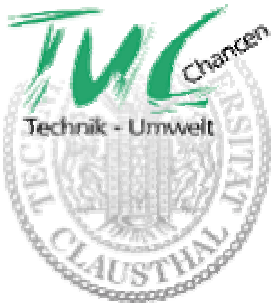

A Framework for Mobile Agents in Peer-to-Peer Networks - Design and Implementation -

Department Business Informatics

Authors: Jorge Marx Gómez und Daniel Lübke

Herausgeber

Prof. Dr. H.-P. Beck
Institut für Elektrische Energietechnik
beck@iee.tu-clausthal.de

Prof. Dr. U. Bracht
Institut für Maschinelle Anlagentechnik und Betriebsfestigkeit
bracht@imab.tu-clausthal.de

Prof. Dr. K. Ecker
Institut für Informatik
ecker@informatik.tu-clausthal.de

Prof. Dr. P. F. Elzer
Institut für Prozess- und Produktionsleittechnik
elzer@ipp.tu-clausthal.de

Prof. Dr. U. Konigorski
Institut für Elektrische Informationstechnik
koni@iei.tu-clausthal.de

Prof. Dr. I. Kupka
Institut für Informatik
kupka@informatik.tu-clausthal.de

Dr. G. Lange
Rechenzentrum
lange@rz.tu-clausthal.de

Prof. Dr. W. Lex
Institut für Informatik
lex@informatik.tu-clausthal.de

Prof. Dr.-Ing. N. Müller
Institut für Maschinenwesen
mueller@imw.tu-clausthal.de

Prof. Dr. H. Richter
Institut für Informatik
richter@informatik.tu-clausthal.de

Managing Editor: Dipl.-Inf. Alexander Hasenfuß

Editorial

Informationstechnik bildet die Basis systemtechnischer Innovationen in Industrie, Wirtschaft und Wissenschaft, aber auch in fast allen anderen Bereichen des öffentlichen und privaten Lebens. Um die hierfür erforderlichen vielfältigen Informationstechnologien in Lehre und Forschung bedarfsgerecht zu integrieren, haben sich Wissenschaftler der technischen Universität Clausthal über Fakultätsgrenzen hinweg zum *Informationstechnischen Zentrum* zusammengeschlossen.

Durch enge Zusammenarbeit des Informationstechnischen Zentrums mit den natur- und ingenieurwissenschaftlichen Instituten stellt die TU Clausthal der Industrie das heute so notwendige fächerübergreifende Expertenwissen für die Erarbeitung komplexer, prozessintegrierter Systemlösungen in einem weit gefächerten Gebiet zur Verfügung.

Ziel der Berichte des Informationstechnischen Zentrums (*ITZ Berichte*) ist es, Beiträge über wissenschaftliche Forschungsergebnisse und Entwicklungen vornehmlich aus Instituten der TU Clausthal aus Bereichen wie formale Systeme der Informatik, wissensbasierte Systeme, Bildverarbeitung und -analyse, Mehrrechnersysteme, Computer Aided Engineering, Mess- und Automatisierungstechnik, Elektrotechnik, Prozessleittechnik zu veröffentlichen und damit einer breiteren Öffentlichkeit zugänglich zu machen. Insbesondere sind dies

- Forschungsarbeiten und Berichte über Projekte, Entwicklungen, Fallstudien,
- eingeladene Beiträge von außerhalb,
- Tagungs- und Workshopberichte,
- Dissertationen.

Bei der Veröffentlichung von Beiträgen wird auf ein zeitaufwendiges Referentensystem weitgehend verzichtet, um Forschungsergebnisse mit minimaler Verzögerung und dementsprechend hoher Aktualität herausbringen zu können.

Die Herausgeber

Preface

This work deals with the requirements, conception and implementation of a framework, which provides a platform and starting point for the development of a peer-to-peer network empowered by mobile agents. Peer-to-Peer networks like Napster, Kazaa etc. have reached a widespread use in the Internet. The idea is to deploy mobile agents, which can travel between network nodes, to a large Peer-to-Peer network. Doing this people could use the peer-to-peer technology for all kind of services like anonymizing network traffic, distributed storage of documents, for replicating contents of heavily accessed Internet sites, trading of information etc. For many of these things there are solutions available but by using a common framework there might be the opportunity to access all kinds of information through a common application programming interface which guarantees extensibility and widespread use. Mobile agents can be used to plug in new functionality into the system without forcing upgrades on users' computers thereby allowing new applications to be deployed immediately over the network. This project has been conducted in collaboration between the Universitat Autònoma de Barcelona (Spain) and the Technical University of Clausthal (Department Business Informatics).

Das vorliegende Werk behandelt die Entwicklung und Implementierung eines Frameworks für Peer-to-Peer-Netzwerke, in denen mobile Agenten eingesetzt werden können. Peer-to-Peer-Netzwerke - wie Napster, Kazaa etc. - erfreuen sich heutzutage im Internet großer Beliebtheit. Die grundlegende Idee ist es nun solche Systeme um mobile Agenten, die zwischen Rechnerknoten wandern können, zu erweitern, um so beliebig und flexibel neue Funktionalität in das Netzwerk einbringen zu können, ohne dabei die Software auf den einzelnen Rechnerknoten aktualisieren zu müssen. Diese Agenten können dann Dienste, wie z.B. Anonymisierung, verteilte Datenhaltung, um häufig benutzte Daten zu replizieren etc., anbieten. Für einzelne Dienste gibt es bereits heutzutage Insellösungen. Durch den Einsatz eines gemeinsamen Frameworks können diese Dienste über gemeinsame und genormte Schnittstellen angeboten werden, welche die weitere Verbreitung und den leichteren Einsatz unterstützen. Das Projekt wurde in Zusammenarbeit zwischen der Universitat Autònoma de Barcelona (Spain) und der TU Clausthal (Institut für Informatik, Arbeitsgruppe Wirtschaftsinformatik) durchgeführt.

Die Autoren sind jederzeit für Kritik und Anregungen dankbar. Diese können per E-mail an folgende Adressen: gomez@in.tu-clausthal.de oder auch daniel.luebke@tu-clausthal.de gerichtet werden.

Jorge Marx Gómez Daniel Lübke

Clausthal, im März 2004

Table of Contents

Index of Abbreviations and Acronyms

ACCC	Academic Computing and Communications Center
ACL	Agent Communication Language
AOL	America Online
ASCII	American Standard Code for Information Interchange
API	Application Programming Interface
CA	Certification Authority
CD	Compact Disc
DFN	Deutsches Forschungsnetz (German Research Network)
DNS	Domain Name System
DoS	Denial-of-Service (attack)
DSA	Digital Signature Algorithm
FCIT	Florida Center for Instructional Technology
FIPA	Foundation for Intelligent Physical Agents
GPG	GNU Privacy Guard
GPL	General Public License
GNU	recursive for “GNU is Not Unix”
GnuPG	GNU Privacy Guard
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines™
ID	Identification
IDE	Integrated Development Environment
IP	Internet Protocol
ISO	International Organization for Standardisation
JAAS	Java™ Authentication and Authorization Services
JAR	Java™ Archive
JDK	Java™ Development Kit
JRE	Java™ Runtime Environment
JXTA	Juxtapose
LOC	Lines of Code
MARISM-A	Architecture for Mobile Agents with Recursive Itinerary and Secure Migration
MD5	Message Digest Version 5
MSN	Microsoft Network
OSI	Open Systems Interconnection
P2P	Peer-to-Peer
PGP	Pretty Good Privacy
PIN	Personal Identification Number
RFC	Request for Comments
RMI	Remote Method Invocation
ROM	Read Only Memory
SHA	Secure Hash Algorithm
SSL	Secure Socket Layer
STDIN	Standard Input
STDOUT	Standard Output
TLS	Transport Layer Security
UML	Unified Modelling Language
UMTS	Universal Mobile Telecommunications System
URL	Unified Resource Locator
UTF	Unicode Type Format
WWW	World Wide Web

Index of Images

Index of Tables

1 Introduction

1.1 Motivation

Peer-to-Peer-Networks usage has been increasing in the last years. Especially file-sharing networks like Kazaa, online chats like ICQ and distributed computing networks like seti@home have become increasingly popular. Today, more than 2.7 million people are using the Kazaa network, which is only one of the available peer-to-peer applications (see Redshift Research (2003)). Peer-to-Peer-Networks are so popular because they allow the easy aggregation of distributed resources like storage space as well as processing power. Additionally modern clients are very user-friendly and due to their open nature, they allow everyone to join the network and benefit from it. The principle of these networks is quite easy: when everyone shares, all will profit.

However, peer-to-peer-networks so far are dependent on their own protocol and their own clients. Thus, no network exists, where all users can join and use all kinds of services like Instant Messaging or File Sharing. Furthermore, if users today want to use a specific functionality, they have to install a new client; the same is valid for each new software- or network-update. This includes, that the systems are not interoperable: For example users of ICQ cannot directly communicate with users of the MSN Messenger although both applications are instant messaging systems.

To avoid this problem, it is necessary to take care of deploying code over a peer-to-peer-network. This should be done in such a way, that the traditional client becomes a facility to accept code, execute it and manage the network's functionality, like browsing resources, keep connections to peers and provide basic message passing. In order to be successful, this client has to be easy to use and easy to install. Therefore solutions, where complete web servers are used to process requests, like LOO suggested (see Loo (2003)), are not satisfying.

1.2 Problem definition and objectives

The challenge today is to bring the required logic into one unified peer-to-peer-network. Mobile code as well as mobile agents is a good solution to address this problem. But at this point there are neither research results nor implementations available which address all issues. This especially includes security, performance and scalability.

Therefore, the long-term goal is to build a peer-to-peer-network whose protocol is scalable and completely decentralized: No central name- or resource-services should exist, because they can be used to control the network, monitor traffic or expose other kinds of control over the network. Instead, users shall be able to send broadcasts through the network for finding resources etc. To bring the functionality into the system the approach of mobile agents, as outlined by LÜBKE AND MARX GÓMEZ (see Lübke (2003)), is used: The peer-to-peer client shall only provide a network component, a user-interface for monitoring the application and initiate requests into the network and a runtime environment, in which mobile agents can be executed.

Mobile agents are transferable pieces of software which can solve tasks and travel through the network autonomously. Normally they are deployed in so called sea-of-data applications, in which data is distributed over lots of systems or may not leave the systems due to privacy concerns or other security restrictions, like medical data in hospitals. Research to such systems has been done intensively, but only under the premise that central security policies and control can be established. An example for such architectures is MARISM-A[1]. However, a public peer-to-peer-network is a different scenario, requiring other security strategies and leading to other problems. But mobile agents are a good solution for adding functionality, because new functionality can be introduced by simply writing and deploying new mobile agents, which will roam the peer-to-peer network searching for specific data or doing calculations on the user's behalf. Furthermore, agents may roam the network while the user's peer is offline, making it very attractive for emerging technologies like UMTS, with much higher connection fees than with traditional connection types.

All in all, this means that the peer-to-peer-network and its protocol are not restricted to a single use-case. Instead, all kinds of applications, from instant messaging over distributed computing to file sharing and collaboration software can utilize the new network infrastructure. Possibly, new applications will be created, we do not know of today.

This work aims to provide the first necessary step towards this kind of peer-to-peer-network empowered by mobile agents:

The first objective is to provide a framework for general peer-to-peer applications using mobile agents. The framework captures the design, which was developed during this project and introduces basic security concepts, specifically designed for the use in open, untrusted networks. To be used for developing, testing and profiling certain parts of a peer-to-peer-network implementation individually, the different functionality domains

[1] MARISM-A can be found in the WWW under http://www.marisma.org

shall be loosely coupled[2]. This way it is possible to design a networking protocol and test how it performs without touching or interfering with other parts of the application like agent-development.

To make further development easier, the framework has to be intensively documented and must provide basic implementations and examples. The basic implementations should provide a starting point for improving the system, so that programmers can concentrate on their part without needing to care about others. However, it is not possible to provide an implementation which is suitable for all kind of real-world applications. An analysis of the strength and drawbacks of the given approaches and implementation suggestions will be made during this work.

Besides the software-design, the second objective is to develop an integrated security-concept: It is based on a decentralised trust-model, allowing each user to establish his security policies as well as supporting the development of anonymous applications. The utilized standard is OpenPGP which has been used in e-mail communications for a long time. Some extensions have to be made which are implemented as simple add-on files transferred with the agents, thus not breaking the standard but providing the needed functionality.

1.3 Structure

This document is structured as follows:

After this introduction, which discusses the goals and the structure, the second chapter introduces and defines all expressions and terms used in this document. It is meant to define all vocabulary used and explain the base-techniques required to understand the overall concept and the involved technology.

The third chapter describes the security model and transport standard developed to secure the use of mobile agents and provide authentication for the agents.

The fourth chapter provides a short overview of the development model used and thus, how design decisions were made. Its purpose is to illustrate which way and for which purpose some designs were favoured over others and how the framework was created. The following chapters all describe a certain step in the development and thus the results for specific sub-processes.

[2] Loosely coupled means that there are as few as possible dependencies between the system's components

In chapter five the requirements for the framework are illustrated. These requirements will be important for the development of the framework.

In the sixth chapter the framework itself is described. This includes an overview about the framework, a specific view into certain pieces and a description and discussion of the basic implementation as a part of the framework.

The seventh chapter gives implementation suggestions for specific problems thus showing the general concept of implementing applications by using the framework and the flexibility offered by it.

The eighth chapter presents a sample application for anonymizing web-traffic by using mobile agents for implementing the networking-logic. The sample application is provided as a proof-of-concept for both the framework's design and the basic implementation provided with it.

Finally, conclusions as well as an outlook of what topics might be interesting for future research, are given in the last chapter.

2 Definitions

2.1 Terms of Object-Oriented Programming

An object-oriented system is composed of objects. An object consists of both data and methods. Methods are the actual operations which can perform actions on the data which are representing the object's internal state. In a well designed system the internal state can only be modified by calling methods. The internal state including the implementation details is unknown to the caller. This principle is called encapsulation and is important for decoupling the system. Internal algorithms and data structures as well as data storage can be easily changed when the caller treats an object like a black box and does not rely on a specific implementation. This led to the principle “program to an interface” (see Magee (2003)) where a programmer is not allowed to make any assumptions about implementation details.

The methods are identified by their signature. The signature consists of a method's name, the method’s parameters’ types, and the method’s return value. Notable is that most programming languages are stricter: the method's name and its parameter must be unique; the return value is often not part of the signature. For instance, in Java™ following two methods cannot belong to the same object, because they have the same Java™-signature:

- `public void setValue(int value);`
- `public boolean setValue(int value);`

However following methods are possible, because their parameter types are different:

- `public void setValue(int value);`
- `public void setValue(String value);`

If a method has the same name (`setValue`) but different parameter types, the method is called an overloaded method. All signatures of an object are called the object's interface.

A type is a name for a particular interface. An object `A` has a special type `T` if all method signatures of `T` are contained in `A`'s interface. For example an object representing a student is likely to have all methods which are required by the type person.

By strictly referring the object's interfaces the black box usage is mandatory because the internal representation, e.g. the data, should not be part of an interface. If methods of an interface are called, it's up to the object's implementation how these requests are served.

The implementation of any object is defined in its class. The class specifies all interfaces it adheres to and implements all methods as well as internal data structures.

An object is created by instantiating a class. An object might therefore also be called in instance of a class. Only by instantiating an object any resources like memory will be allocated. There may be as many objects of a class as the local resources permit. Each of these objects has its own internal state. That means each object has its own set of variables and is not influenced by methods executed on other objects.

Classes can be inherited from other classes. If class A inherits from class B, class A is also called a subclass of class B. Both classes share the same implementation of the methods, however the subclass may override a method or add new methods but may not drop any methods. Overriding means that the subclass specifies a different implementation for a method.

Whenever this method is called, the overridden implementation will be executed, regardless of the interface through which this call was conducted. This behaviour is called polymorphism and provides a lot of flexibility to the software designer.

A special case of an class is the abstract class. It is a class which implements an interface only partly but specifies method signatures which have to be implemented by subclasses. Because the implementation is not complete, no object may be instantiated from an abstract class. Abstract classes are often used to provide an interface as well as a default or common implementation for the classes which are using it.

2.2 Design Patterns

A pattern definition used for patterns in towns and buildings, but also matches the software patterns comes from CHRISTOPHER ALEXANDER: “Each pattern describes a problem which occurs over and over again in our environment and then describes the core of the solution of that problem, in such a way that you can use this solution a million times over, without oever doing it the same way twice.“ (see Gamma (2001), p. 2)

Since “designing software is hard and designing reusable object-oriented software is even harder“ (see Gamma (2001), p. 1) people are trying to further simplify the software design process.

Many problems are reoccurring quite often in different facets. Consequently the idea was born to abstract the common problems and nail them down to their important properties, like design of hierarchical systems, object creation etc. as was done within archi-

tecture long before. Solutions are then provided for these common problems which have been proven in existing software designs and projects.

A formal definition of a design pattern in the software design process is: “A pattern is the abstraction from a concrete form which keeps recurring in specific non-arbitrary contexts.” (Riehle (1996))

A pattern description normally consists of

- pattern name,
- problem description,
- proposed solution, and
- consequences for the design.

It is important that design patterns deal with how objects should generally behave within a given problem context and not to provide answers for implementation problems like, for example, linked lists. (see Gamma (2001), pp. 3)

The standard reference for design patterns was written by ERICH GAMMA, RICHARD HELM, RALPH JOHNSON and JOHN VLISSIDES and published in their book “Design Patterns“. But especially in the last years there were more publications of design pattern catalogues (see Cooper (1998), Grand (1998), Microsoft (2003), Monday (2001), Zorgdrager (2001)) and often new patterns and their application are discussed in programming related magazines and publications (for example Horton (2003).

The advantage for programmers and software designers alike is that by referring to standard names for standard design elements communication and code clarity can be improved. Moreover beginner software architects have a good point for searching initial solutions geared towards their problems.

Although design patterns are often used today, some people have expressed criticism. A main point is the so called “pattern trap” which says that it is likely that (inexperienced) software designers will not focus on solving a problem in the best way possible anymore but instead are trying to use as many design patterns as available to have an apparently good design.

M. J. DOMINUS has another remark: In a short presentation, which he published afterwards on the web, he expresses that “the design patterns solution is to turn the programmer into a fancy macro processor” (see Dominus (2002)) because only design tem-

plates are used over and over again and take lots of creativity from the coder. Also many design patterns would address problems which are existent only in old ("1970s-era") languages and are solved within newer languages.

2.3 Frameworks

A framework consists of interfaces, abstract classes and classes which together make up a reusable design for a specific class of problems (see Deutsch (1989) and Johnson (1988)).

The decomposition of the problem is done once and the resulting design expressed and implemented in interfaces and classes, can be reused to save time in further projects. These projects are realized by customizing these frameworks by implementing and sub-classing interfaces and abstract classes and plug these into the framework to form a whole application. For this, frameworks utilize the polymorphism of object-oriented languages. The implementation is provided by the programmers' subclasses although the framework executes methods which are required by interfaces and classes within the framework itself.

Frameworks are available for lots of problems like JXTA (see Project JXTA (2003)) for developing peer-to-peer-applications or D'Agents (see Darthmouth (2003)) for developing agent-based systems.

In contrast to toolkits, frameworks capture the design of an application[3] while toolkits capture common functionality and implementation like lists, file or database access. Toolkits are very common and at least one is usually shipped with the common compilers and development environments, like the GNU C-Library (see GNU (2003)) for most C Compilers or the Visual Component Library for Delphi (see Watson (2003)).

Both design patterns and frameworks try to capture design decisions, document them and make them reusable for later use. The main difference between both is the scope they have: While frameworks capture an entire design of an application, design patterns concentrate of a specific design decision. A framework normally consists of thousands of these decisions and often uses solutions which are propagated by design patterns to solve them.

This has some implications: Design patterns are much more general and therefore more abstract. Because of this, only examples of design patterns can be embodied in code

[3] The application-design comprises all interfaces, abstract classes and classes and their structure of a given application

while frameworks are written and shipped readily in a programming language (see Gamma (2001), p. 28).

The advantages of using a framework include:

- Faster development time: Because the software designers and programmers can concentrate on the specifics of the application and do not need to care about the design because many if not all design decisions have already been done.

- Similar structures of similar applications: because the framework dictates the design all applications which use this framework will automatically have the same or at least a very similar design and probably use the same notion within the source and the documentation. This makes it easier to maintain these applications and to further utilize the skills of programmers and software designers.

However there are some drawbacks one should be aware of before choosing a framework:

- Lost control: the control flow is defined by the framework and cannot or only very expensively be altered within projects.

- Changes in the framework may lead to expensive changes in the software: because the frameworks provides the software design and defines the control flow of the software, nearly all changes in the framework lead to updates within the software.

Frameworks have become more common and important in today's software industry because they significantly reduce time-to-market and cut down costs.

2.4 Agent

There are lots of different definitions what an agent is. Within this context we assume that an agent is a process that is able to autonomously initiate changes within its environment and react to changes therein (see Tanenbaum (2003), pp. 202). This definition matches a lot of processes but contains the very important characteristics which distinguish agent-bases systems from "standard" software. These characteristics are summarized in table 2-1.

Property	Description
Autonomous	Can act independently, e.g. with no intervention of users
Reactive	Reacts in time to changes in its environment
Proactive	Initiates actions which are changing the agent's environment
Communicative	Can exchange information with other systems, including but not limited to users and other agents

Tab. 2-1 Characteristics of an agent (see Tanenbaum (2003), pp. 203)

The Foundation for Intelligent Physical Agents (FIPA, see FIPA (2003)), a non-profit organisation aimed at producing standards for the interoperation of heterogeneous software agents, has released lots of specifications dealing with agent-based systems. A proposed architecture of a system for mobile agents can be found on their website (see FIPA (2002)). The FIPA specifications mostly deal with inter-agent communication. Today, most specifications deal with messaging-standards. Therefore, it is very difficult to implement all of the FIPA messaging standards, because they are very generic and powerful, although under most circumstances only subsets of the functionality is needed.

All agents that want to be able to communicate with each other need to agree on a common language. A standard for such a language is the FIPA Agent Communication Language (ACL, see FIPA (2002a)) which specifies a standard language for inter-agent communication.

To further describe the function of an agent there are subclasses like Collaborative Agents Interface-Agents, Information-Agents and Mobile Agents; the latter is the important one for the creation of this framework.

The use of software agents is also propagated as a solution to common software engineering problems (see Jennings (2000)).

2.5 Mobile Code

In classical client/server- and multi-tier- applications only data will be sent over the network. However it may be beneficial or necessary to transfer code and execute it on another computer. This process is called migration. Advantages of using mobile code are (see Tanenbaum (2003), p .186):

- Migrate processes to systems which have unused resources available to distribute the load.

- Execute code near the data or the input, which means that less data needs to be transferred over the network thus enhancing response times.

However sending and executing code on other machines raises some concerns:

- The source of the code needs to be authenticated or executed in a secure environment so that no damages to local data and/or other resources may occur.
- The migration possibly shall take place between different platforms which means further efforts have to be made to make the code portable.

There are many possibilities for mobility. To further categorize mobility there are three possible characteristics of mobile code:

- Weak vs. Strong Mobility: In a system which uses weak mobility only the code is transferred between two systems, possibly including initialization data. In contrast, strong mobility means that a process may be stopped at any time and then the code as well as the current state is transferred to another system, where the execution proceeds as if no break would have occurred.
- Sender vs. Receiver initiated mobility: The type of mobility categorizes between who took action to initiate the code migration. Either the receiver requested code from a sender which is transferred and then executed, e.g. a Java™ applet, or the sender wants to migrate a piece of code to another machine like distributed search programs.
- Execution in the same or a separate process: The receiving system has to decide whether the received mobile code has to be executed in the same process or whether a new process should be spawned. Utilizing the same process is easier to implement and conserves resources but the process has to take security measures. By creating a new process the operating system is responsible for security. For this reason a safe implementation might be easier.

Mobile code can be an effective way of implementing applications. However the programmers need to take care of security.

2.6 Mobile Agent

A mobile agent is a special type of agent which is capable of travelling between different computer systems.

Mobile agents are implemented using mobile code. That means agents can migrate between different machines. By deploying mobile agents the agents' requests can be issued near the target and the responses do not need to travel across the whole network which can improve the response time. However, by utilising mobile code, mobile agents not only share the advantages but share the common problems as well which are espe-

cially security and platform independency. A main task of this framework is to provide solutions to these problems.

A mobile agent normally uses weak mobility that means it carries his state from device to device till his purpose has been fulfilled, but normally the current execution state, for example the stack, cannot be migrated. The migration is normally initiated by the sender as well, e.g. the agent itself (acting autonomously) or the system it runs on (for example before it is shutdown).

2.7 Peer-to-Peer-Network

The term peer-to-peer-network (p2p-network) has a wide range of definitions that are nearly identical but sometimes have very different emphasises.

Common to all definitions is the fact that a p2p-network is a network of devices which are able to access resources on all other devices as well as providing resources to them. The main issue where most definitions significantly differ at is the question whether this network may have some central services like servers[4] for special supporting purposes, especially name resolution, or not[5].

Examples for p2p-networks include ICQ (instant messaging & chat, see ICQ (2003)) and eDonkey (file sharing, see eDonkey (2003)) for p2p-networks with centralized lookup-services on the one hand and Windows Networking (file and print services, see WOWN (2003)) and Gnutella (file sharing, see Limewire (2003)) without centralized resources on the other hand.

The different implementation of how to resolve names shows a real problem when designing a p2p-network: Because nodes can join and leave and may rejoin with different IP addresses and consequently with different DNS names, a p2p-network needs to design and manage its own namespace. This resource centric addressing, like your chat nickname, is something which may be seen as one of the greatest changes and perhaps benefits when using p2p-networks (see Shirky (2000)).

Within the last years p2p-technology has become generally known through the rise of file-sharing software. This software is generally used to swap copyrighted material and is therefore illegal. Therefore many people think that p2p-networks can only be used for copyright infringement, although there are many new developments like a distributed search engine DFN S2S (see DFN (2003)) designed by Germany's National Research and Education Network (Deutsches Forschungsnetz, DFN).

[4] see Fontana (2002) for an example
[5] see Foster (2003) for an example

3 OpenPGP-Transport Security

3.1 Requirements for a security concept

Communication over untrusted connections, like the Internet, need to be secured in order to protect data and privacy. Normally, security is established by authenticating connections and encrypting the data between them. However, in p2p-networks, in which anonymous users can log in as well as users can authenticate themselves, the problem becomes more difficult. The problem is even more complicated since each network host must assign access rights to mobile agents.

A new security-model for distributing agents had to be developed, since the standards in use do not meet all of the following requirements for open and freely usable p2p-networks:

- if required, provide authentication for agents, agents' owners and agents' programmers in an open p2p-network,
- if required, allow anonymous usage of the network,
- do not use any central resource,
- provide secrecy in terms of encryption for transmitted data, and
- allow agents to have different security-settings on different hosts.

The introduced security-concept, named OpenPGP-Transport Security, addresses all of these problems. But due to the nature of mobile agents and open networks, it is not possible to rule out all security problems, especially the malicious host problem, which has not been solved today (see Bierman (2002) and Marques (2002)). The OpenPGP-Transport Security concept is based on the ideas initially outlined by LÜBKE and GOMÉZ (see Lübke (2004)).

3.2 Introduction to Public-Key Cryptography

For providing confidentiality of transmissions, data need to be encrypted and for authenticating transmissions, data can be digitally signed. For this, public-key cryptography will be used, since it allows the easy distribution of encryption keys without the

need of trusted and secure communication links, which are not available in open p2p-networks.

Public-key cryptography uses two keys per party, one called the public and the other the private key. The public- and private-key are generated in the same process. Theoretically, it is possible to calculate one key if the other is known. But in practice this problem is mathematically so difficult that it is impossible to do this calculation, because it would take millions of years on average (see Johnson (1999)) to do so.

The interesting property of these keys is that they are complementary: Data encrypted with one can only be decrypted with the other and vice versa.

This allows the distribution of the public-key over insecure channels to all entities with whom one wants to communicate. The public-key can then be used by the sending entity to encrypt the data which is send afterwards to the receiver. Since the receiver is the only one who has the private-key, he is the only one who is able to decrypt the data. This process is illustrated in figure 3-1.

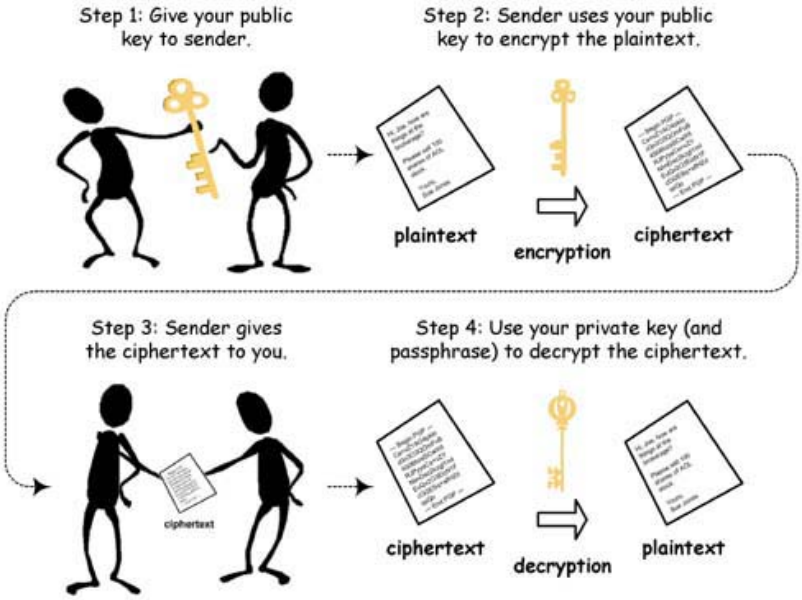

Figure 3-1: Public Key encryption process (ACCC (2000))

For signing data digitally, the sender first calculates the hash-value of the message and encrypts it with his private key. A hash-function is a function which calculates a small fixed-size value from a variable length input. In no way, it is possible to extract any information about the original message from the hash-value. Furthermore, the hash-functions used in cryptology must guarantee, that it is as difficult as possible to calcu-

late two messages with the same hash-value[6] (see Hyperdictionary (2003)). The encrypted hash later is encrypted with the sender's secret and can be decrypted with the sender's public-key. At this point, the recipient of a transmission is able to compare the decrypted hash-value with the hash-value of the transmission. If they match, the digital signature is valid, if not, the data has been changed in transit[7].

Probably the most known algorithm for public-key encryption is RSA[8] - named after its developers RONALD RIVEST, ADI SHAMIR, and LEONARD ADLEMAN. Hash-functions commonly used are MD5[9] (Message Digest Version 5) and SHA[10] (Secure Hash Algorithm).

3.3 Public-Key Cryptography and Trust

However, being able to encrypt something with a public-key does not solve the whole problem. One can exchange the public-keys over insecure media without putting the secret key at risk, but it can not be guaranteed, that someone replaces the originally transmitted key with his own key. Encrypting to the wrong key enables an attacker to read the sensitive data in transit.

Therefore, one needs to trust a key before using it, so that traffic is not encrypted to an attacker, who is also called “man-in-the-middle”. To accomplish this, a validation process has to be established which leads to a public-key infrastructure (PKI) where users and software can retrieve and validate keys.

Many systems, which are designed for deploying mobile agents, like MARISM-A, utilize the X.509 certificate standard for storing and managing keys in the underlying public-key infrastructure, which is defined in RFC 3280 (see Housley (2002)).

X.509 depends on a centralized architecture, in which so called Certification Authorities (CA) issue certificates. The certificate is trusted, if the CA is trusted and the CA has issued the certificate containing the key. Examples for CAs are VeriSign[11] or TC Trust-

---

[6] Because the target set of the hash-function iss maller than the set of the input values, there will always be duplicate hash-values. The question is, how difficult it is to find two inputs which give the same output.
[7] A more complete introduction is given by Davi Youd (Youd (2000))
[8] The original version is (Rivest (1978)), an easier explanation is given under (DI Management (2003))
[9] MD5 is defined in RFC 1321 (Rivest (1992))
[10] SHA is defined in RFC 3174 (Eastlake (2001))
[11] Corporate homepage is http://www.verisign.com

Center[12]. The CA may also sign another entity and allow it to sign other keys as well. This chain of CAs and the key is called a trust path or certification hierarchy.

The public-key together with user information like name, e-mail etc., and the complete trust path, including the public-keys, are stored in this so called certificate. The certificate may also contain information about what a user is allowed to do. These may be technical, like acting as a CA, or from a economical point of view, like ordering goods for a company. However, the X.509 standard is very imprecise in this aspect, which has lead to differing and incompatible implementations how to handle these extensions. This and many other drawbacks of the standard are discussed in the X.509 Style Guide (see Gutman (2000)).

The X.509 standard is widely used in industry for storing keys which are used for securing web transactions (Secure Socket Layer (SSL, see Erkomaa (1998)) and Transport Layer Security (TLS, RFC 2246 (Dierks (1999)) or for sending e-mails (S/MIME, RFC 2633 (Ramsdell (1999)) and is tightly integrated in standard software like browsers and e-mail-software. However, the central approach has some drawbacks:

- if the CA's private key is exposed, the whole security model collapses,
- a key can only be signed by one CA,
- only CAs can sign keys,
- for each CA the certificate has to be installed locally,
- therefore, the CA key has to be transferred securely to the local machine via a secure connection, and
- implementations of peer-to-peer networks without any central resource are not possible.

Because of these drawbacks, especially the last one, this framework defines a security concept without the use of the X.509 standard, which can be used for establishing a security model in agent-driven p2p-networks.

[12] Corporate homepage is http://www.trustcenter.de

3.4 The OpenPGP Standard

In 1991 PHIL ZIMMERMANN released the first version of Pretty Good Privacy (PGP). It was developed to make the use of encryption easy and usable for private persons. It uses strong public-key encryption and digital signatures. Because of this and the legislation in the United States of America during that time, PGP faced lots of legal problems, because it was considered to be a weapon. All these issues are resolved today. PGP was further developed and the program changed ownership a lot. Since 2002 it is owned by PGP Corporation[13].

The original message format was slightly changed and was standardized as RFC 2440 (see Callas (1998)) as OpenPGP in the year 1998. Today it is widely used in the academic and open source area to encrypt and sign e-mail traffic and software packages.

The OpenPGP standard allows the use of many encryption and signing algorithms and allows the extension by any new algorithm. Commonly supported ones for encryption and digital signatures are:

A key should uniquely be identified by its key fingerprint. The fingerprint is a hash value, consisting of 128 bit, which should be unique world-wide.

However, for simplicity, when normally referencing keys, so called key-ids are used, which are the last 4 bytes of the key fingerprint. These are not unique world-wide (see Harris (2002)), but can used practically without causing too many headaches.

Instead of relying on a hierarchical trust model, OpenPGP uses a decentralized approach which is called the “web of trust”. Its principle is very easy: Everyone can sign any key. By signing, one guarantees that the key really belongs to the one, whose name is saved with that key. This practice leads to a graph which represents the trust relationships between the keys and their persons. A real world example is illustrated in figure 3-2.

[13] Corporate homepage is http://www.pgp.com

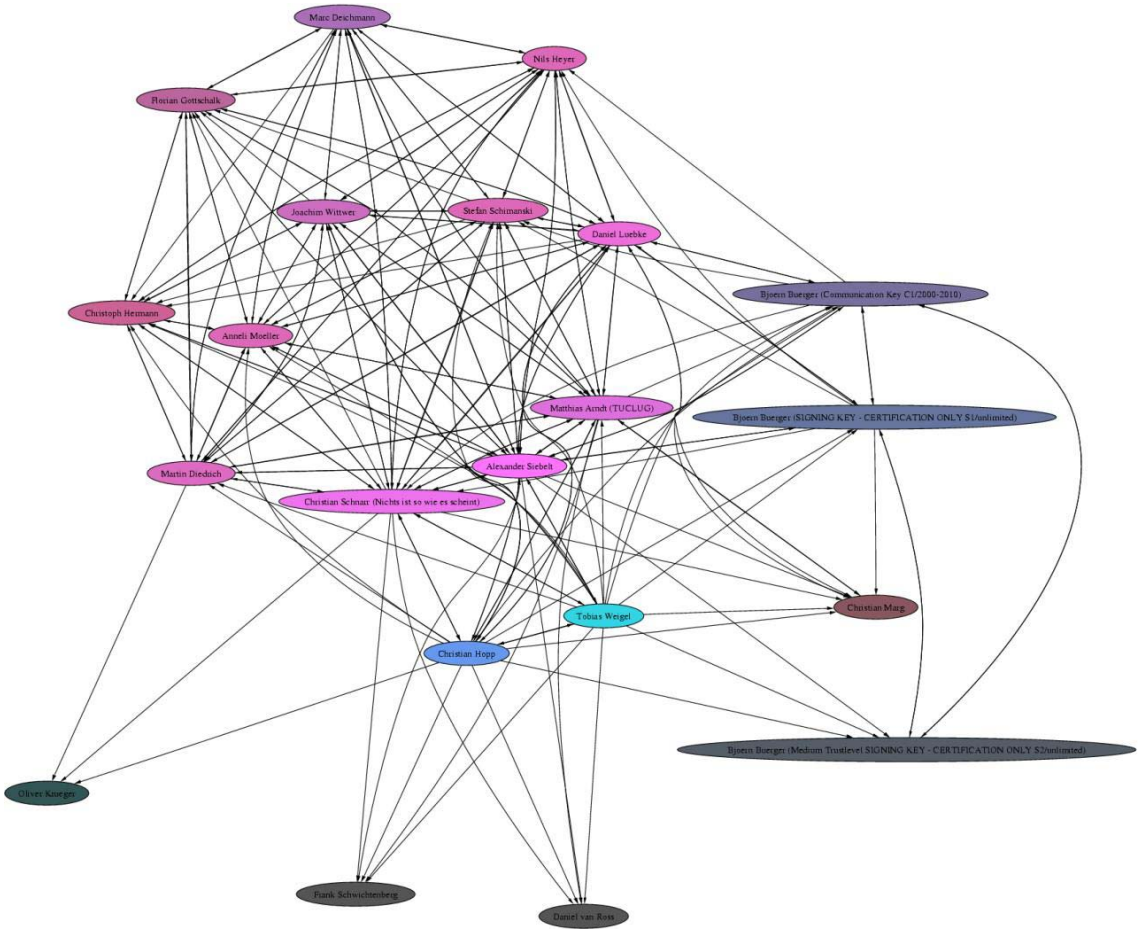

Figure 3-2: Example of a Web of Trust

To validate a key, the user has to obtain a trustworthy chain between his key and the key he wants to use. This chain is called a path or trust-path. For making this task easy, there are so called pathfinders, like Jonathan McDowell's (see McDowell (2002)). To establish these signatures, keys are signed between colleagues, friends and during so called key-signing parties. There are statistics available for the web of trust, as well as for individual keys. One very prominent example is maintained by HARRIS (Harris (2003)).

The main problem with this approach is, that one often has to trust one or many chains between one's own key and another. The trust in a chain can be controlled via so called "owner-trust", which describes to which degree one trusts signatures made by another key: If one thinks, the key's signatures are fully trustworthy, that means that a signature really does guarantee that the user carefully verified the key's ownership, one will assign full owner-trust to that key. If one is not sure, also marginal trust or no trust can be assigned.

This way, it is possible to establish CAs in the OpenPGP world as well. The standard proposes an ultimate owner trust, which normally is only assigned to one's own key.

This ultimate trust can also be assigned to other keys which in turn means, that this key is as trustworthy as one's own key. So by assigning ultimate owner trust to a key, that key becomes a de facto CA key. There are CAs for OpenPGP keys as well, like HeiseCA[14], which can be treated as a CA but can also be treated as a key like any other, depending on the user's trust settings.

However, it is not possible to inherit trust from a CA: With X.509 a CA may certify another CA. If one trusts the top-level CA, one will also trust certificates issued by the lower-level CA, which automatically is not possible in OpenPGP because the user has full control about the trust settings.

The OpenPGP standard is widely implemented. The most important examples are PGP and the GNU Privacy Guard (GnuPG, GNUPG (2003)). GnuPG is an open source development, which was also funded by the German government and ported to a variety of platforms, like Windows, Linux, MacOS X, etc. Both implement the cryptographic algorithms and the standard message format. PGP also allows integration into the most famous, commercially used e-mail clients, while GnuPG uses plug-ins, developed by third parties for integration.

3.5 OpenPGP vs. X.509

Although OpenPGP and X.509 deploy the same cryptographic algorithms and are therefore equally technically secure, both are using a completely different trust model.

This leads to some differences: In the OpenPGP world the user has more control over his security settings. There is not necessarily a CA and keys can be verified by more than one person. In turn, the user has more responsibilities and but also full control of his communication.

The CA in X.509 public-key infrastructures represents a single point of failure, but allows easy key distribution, because on the clients no additional trust-settings have to be made. Failures might be technical, e.g. leak of the CA’s private key, or social, e.g. false certificates are issued.

Another advantage of CAs is, that a certificate is either fully trusted or not trusted at all, depending whether the CA is trustworthy or not and the certificate is valid or not. In OpenPGP, keys can be marginally trusted because of the paths through which the keys

[14] Project homepage is http://www.heise.de/security/dienste/pgp/

are being validated. There might not always be a short path between two keys making the decision if a key is trustworthy or not very difficult.

An advantage of the OpenPGP model is that the key-validation is not commercialized. CAs are normally getting paid for issuing certificates, but the key-signing process with OpenPGP is free.

Furthermore, the approach of the web of trust is more suited to the world of p2p-networks: No central resources are needed and the security infrastructure is as easily extensible as the network itself. Each user can choose his individual security settings in a way similar to how he chooses to share files today. Anonymous users can use the p2p-network, as well, and users do not need to pay for official certificates. This allows more users to use the p2p-network, encouraging further usage and bringing more resource to the network.

For both X.509 certificates and OpenPGP implementations are available, in commercial or free software, so that they can be easily be integrated into applications. Both systems have proved their strength in day to day applications and are the only remaining standards in cryptography, which are used in practice.

3.6 OpenPGP and Mobile Agents

The OpenPGP standard is well suited for use in p2p-networks. It allows a security model which is not dependent on central resources like CAs. Instead, users can choose and verify which keys and users are trustworthy.

Furthermore, bringing the web of trust to the world of mobile agents, new applications are possible: applications in which agents can learn trust, p2p-networks of agents which are operating in an open manner etc.

The question, which therefore needs to be resolved, is what parts need to be encrypted or signed and on which things security policies can work and decide which rights an agent has on the system or within a transaction.

The proposed security architecture is based on two signatures for the agent and three keys: One signature for the code, which identifies the programmer of the agent. The next signature identifies the agent's owner. Furthermore, hosts have to encrypt and sign the agent's state when it is sent over the network.

The permissions an agent has within transactions, like on-line auctions, are stored within an agent passport, in which the owner can state which limits an agent has.

Every participant in the p2p-network can have a key: a host, a user and an agent. The keys are correspondingly named host-key, user-key and agent-key.

The host-keys and agent-keys are identified by a prefix in their description: “HOST:” and “AGENT:”, like “HOST: myhost”. It should be taken care that these two key types are never uploaded to the central key-servers. Instead their distribution should be part of the p2p-network, so that the central key servers are not polluted by keys belonging to virtual entities, which are not part of normal e-mail communication. The user-keys can be uploaded to the key-servers and can be used for normal e-mail traffic as well.

3.7 Packaging-Format

The OpenPGP-Transport format packages all agent-related information into a ZIP archive. The ZIP archive provides two main advantages:

- The ZIP archive is compressed thus reducing the used network bandwidth for sending agents, and
- the ZIP archive can contain multiple files, so that all information can be stored into it, but only one file needs to be transferred.

Inside the ZIP archive there are following items:

- agent.jar: Contains the agent's code inside a Java™ archive (see Hyperdictionary (2003)). The Main-class property of the Java™ archive has to be set to a class implementing the agent's logic.
- owner.asc, programmer.asc, agent.asc: Contain the owner-key, programmer-key and agent-key in OpenPGP standard format. The files can be a binary representation of the keys, although ASCII-armoured representation is preferred.
- ownercert.asc, programmercert.asc: Contain the verification of the owner and the programmer.
- passport.txt: Contains the clear signed agent-passport as described below.
- state.dat: Contains the serialized state of the agent.

To differentiate between hosts, there can be alternate versions of these files. They have to be named file.host.suf, like passport.myhost.txt. This indicates that the host myhost should use this agent-passport which is encrypted using the myhost's public host key.

3.8 Keys

OpenPGP keys are used to guarantee the safety in this security concept. There are three public-keys of interest for the security system, one for each assigned party to an agent:

- Agent-Key
  The agent-key is the agent's own key. The corresponding key pair is generated before deploying the agent and is unique to this agent's instance. Using the secret key on the owner's peer, it is possible to sign data for the agent, which it can verify on selected or each target peer using its public-key. This way, it is possible to implement the itinerary protection scheme initially developed for the MARISM-A platform. The agent's key is stored inside the ZIP archive as agent.asc.

- Owner-Key
  An agent does actions on behalf of its user without the user needing to oversee its actions. However, security settings may vary for agents of different users, like hiding personal data and opening them only for friends. Therefore, agent owner's must be reliably identified. This can be accomplished by signing the code and certifying the agent was sent by a specific user. Furthermore data can be encrypted to the owner only, thus guaranteeing that the accessed data are really only readable by him. The owner's key is stored inside the zip-file as owner.asc.

- Programmer-Key
  Sometimes it can be useful to trust a programmer. For example if an anonymous agent visits a peer, it has no owner associated but rights have to be granted or not. If there are some standard types of agents they can be certified this way, allowing them to enter the system with appropriate security settings in effect. The use of the programmer-key is somewhat similar to the trust model of ActiveX (see Microsoft (2001)). However, the user can limit the agent after it is accepted by the system by using the Java™ security model instead relying completely on this certificate. The programmer-key is stored inside the ZIP-file as programmer.asc.

Note, that all keys can be present for an agent, only some or even none. The peer has to choose the security-settings and must be aware of this.

3.9 Agent Passport

The agent-passport, which is stored as passport.txt, is a clear-signed[15] name/value-list, like in the following example:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

AgentName=ExampleAgent
Owner=NoOwner
OwnerHost=SomeHost.domain.com
Expires=1900-01-01 00:00
-----BEGIN PGP SIGNATURE-----
...
-----END PGP SIGNATURE-----
```

Values are simply stored as name-value pairs, where the name and the value are both case-sensitive, e.g. the attribute NAME is not equal to the attribute Name. The whole name-value list has to be signed by the owner's key. The signature has to be a clear-signed OpenPGP signature.

There are some standard-fields available, which should be supported by all passport implementations:

- AgentName
  The AgentName directive specifies the name of the agent. This way, the name can be authenticated, because it is contained in the signed data block.

- Owner
  The Owner directive can contain a clear-text name of the owner, which is only for reporting etc.

- AgentKey
  This option specifies the agent's key. That way it is possible to verify, if the agent's key has been exchanged in the transmitted ZIP archive. If the ZIP archive does not contain an agent's key, the system will try to fetch the key otherwise. If this is not possible, the agent must not be accepted.

- OwnerHost
  The OwnerHost option specifies from which peer the agent was started. This peer is considered the agent's home peer, to which the agent can request to go back.

[15] A clear-signed signature combines the signed data and the signature into one document, in contrast to a detachted signature where the signature and the signed data are seperated into two files

- Expires
  A passport can expire at a given date, which has to be specified as YYYY-MM-DD HH:mm, like 2004-01-31 00:00. After this date, the passport may not be used and the system should reject the agent completely.

Note, that any name-value pairs can be put into the passport, so this approach is very extensible. For instance, PINs, passwords, further identification, e-cash etc. can be stored inside the agent's passport.

3.10 Owner Verification

The owner is being verified by an extra owner certificate, which can be transferred within the ZIP archive. It is not mandatory, so if one wants to send an agent anonymously, one can leave out the owner certificate.

An example owner certificate looks like this:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

OWNER_CERTIFICATE

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
-----BEGIN PGP SIGNATURE-----
...
-----END PGP SIGNATURE-----
```

It is a clear-signed plaintext-file whose first line must be OWNER_CERTIFICATE followed by a blank line. After the blank line, the MD5 hash of the agent’s archive is saved. Since the owner certificate is signed by the owner, nobody can tamper with it. However, the owner certificate should only be used in conjunction with a valid agent passport and an agent-key, because otherwise the system is open to replay attacks, where an agent is send to the network which uses the same code but using other instance data. Depending on the agent's implementation this can lead to security risks, so it is necessary to always verify the agent-passport as well and to make sure, that the owner certificate is signed by the same key as the passport is.

3.11 Programmer Verification

The programmer of an agent can be identified the same way as the owner. The certificates for the owner and the programmer are differing only in the first-line certificate identification string which is PROGRAMMER_CERTIFICATE for this type of certificate, as can be seen in the example:

```
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA1

PROGRAMMER_CERTIFICATE

aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
-----BEGIN PGP SIGNATURE-----
...
-----END PGP SIGNATURE-----
```

The programmer is the author of the agent code and need not be affiliated with the current agent's owner. The use of the programmer verification process is the ability to allow code developed by trusted third parties to enter the system. For example, a user may decide to only trust a specific software company and may configure this p2p-client to reject all agents which are developed by others.

4 Development Process

The framework has been developed a sequential process like the classical Waterfall Model (see Barkstrom (2000)). The Waterfall Model divides the development process into strictly separate phases which are conducted sequentially. Each phase is dependant on its predecessor so that it is basically a step-by-step approach spanning from the conceptional idea to the final product.

The conceptional idea has already been outlined in the introduction and was the task to solve. The second step consists of the requirements-specification. Since the requirements of the framework include all design requirements of the applications based on it, the next chapter will list the general design requirements, e.g. flexibility, as well as the application use-cases which have to be supported. These use-cases have been selected by looking at existing applications and frameworks and based on the initial p2p-concept.

The next step was to decompose the whole system into different functionality domains. These domains should be as independent from each other as possible and were separated into packages.

The design for each package was then developed, defining the interfaces and specifying their responsibility. During this step, the future control-flow became visible the first time. This lead to some changes in the packages - especially the introduction of the container-package - to where more and more interfaces, related to the execution environment, were moved.

Afterwards, the implementation of the supporting abstract classes and the basic implementation were done. This process was accompanied by small test-fragments, which were used to verify that the implementations were working as expected. However, during this process, it became visible that the system was decomposed into packages, but that too many reference on runtime-objects existed, making the decoupling harder. For example, the OpenPGP-Transport implementation was dependent on the GnuPG cryptographic routines. To resolve this, so called managers were introduced to decouple the system. This leads to some changes in the interfaces, which made them cleaner and resulted in a very flexible and modular design.

After completing the framework, the sample application was developed. The design of this application is described in chapter eight. The application was a proof-of-concept for the design. Although the design worked, the application showed some parts, where probably some functionality would be used by every application based on the frame-

work. For this reason, some convenience routines were moved from the sample application into the framework.

The last step was to verify the javadoc documentation in the source-files, which had been inserted there right from the beginning but was extended at the end to make some parts more precise.

The result is a framework, consisting of approximately 9100 lines of code (LOC), including the javadoc documentation, which has proven in a sample-application that the design is good and usable.

For developing the whole system and the sample application, following software was used:

- Sun Microsystems™ JDK 1.4
- Eclipse 2.1
- ArgoUML/Poseidon Community Edition

The choice of the JDK is obvious: It is the standard implementation of a Java™ VM produced by the inventor of the Java™ platform and should therefore guarantee compliance to the Java™ standard.

Eclipse, as shown in figure 4-1, is an open-source project[16] initiated by IBM™ which aims to provide a framework for an integrated development environment (IDE). Everyone is invited to supply plug-ins to Eclipse which are providing new functionality. The standard plug-ins provide a very good coding environment for Java™ including advanced refactoring methods for rearranging code fragments and changing designs on the fly. For this reasons, Eclipse was chosen for developing the framework since almost no graphical interfaces needed to be developed but much coding was to be done.

[16] The project homepage is http://www.eclipse.org

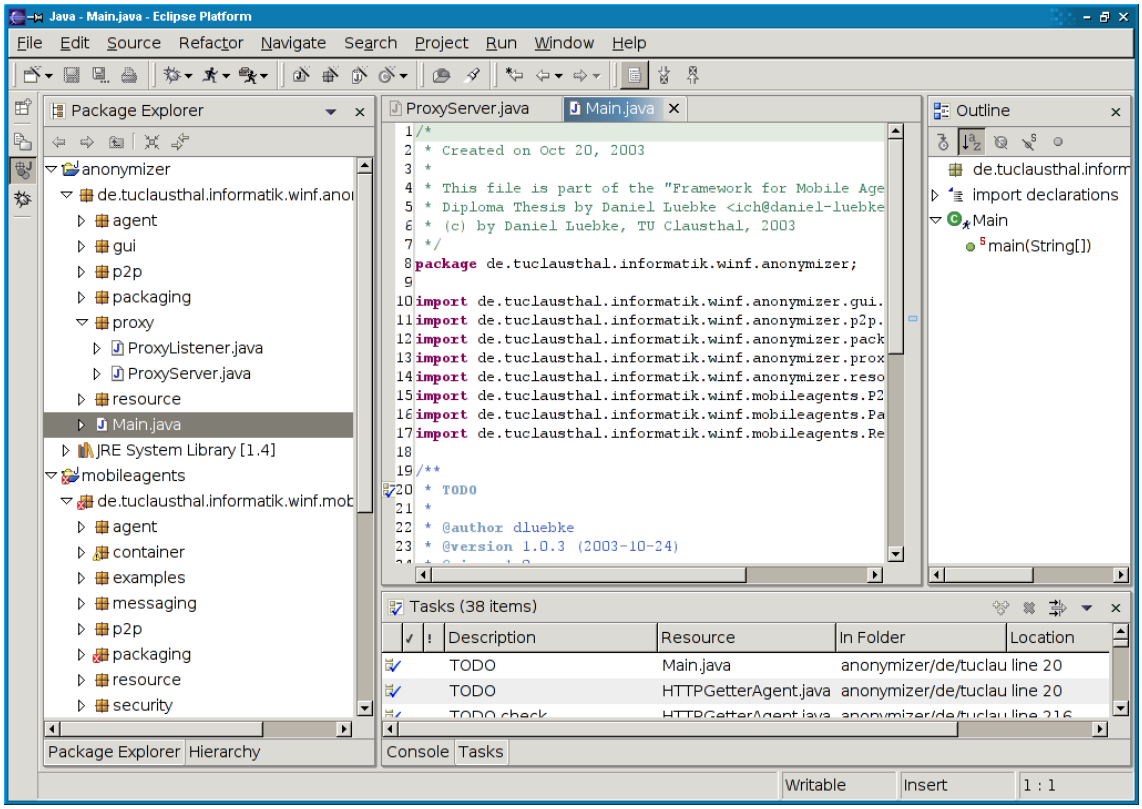

Figure 4-1: Screenshot of the Eclipse IDE

ArgoUML[17] is an open-source design tool, producing UML diagrams. It has a commercial counterpart Poseidon[18], which is capable of producing more types of diagrams and can read ArgoUML's files. Both were used for creating the UML diagrams to document the framework's design.

[17] The project homepage is http://argouml.tigris.org
[18] Tigris', the manufacturer of Poseidon, homepage is http://www.gentleware.com

5 Framework Requirements

For being successful and widely used in as many application types and applications as possible, the framework has to provide a basic set of functionality. These requirements can be associated with at least one category: basic functionality, security, flexibility, and ease of use.

5.1 Basic Functionality

The requirements will be outlined below including typical use cases which shall demonstrate why and for which purpose features are introduced. The use cases can be sorted into one of the categories agent-related, migration-related, resource-related or agent-subsystem-related.

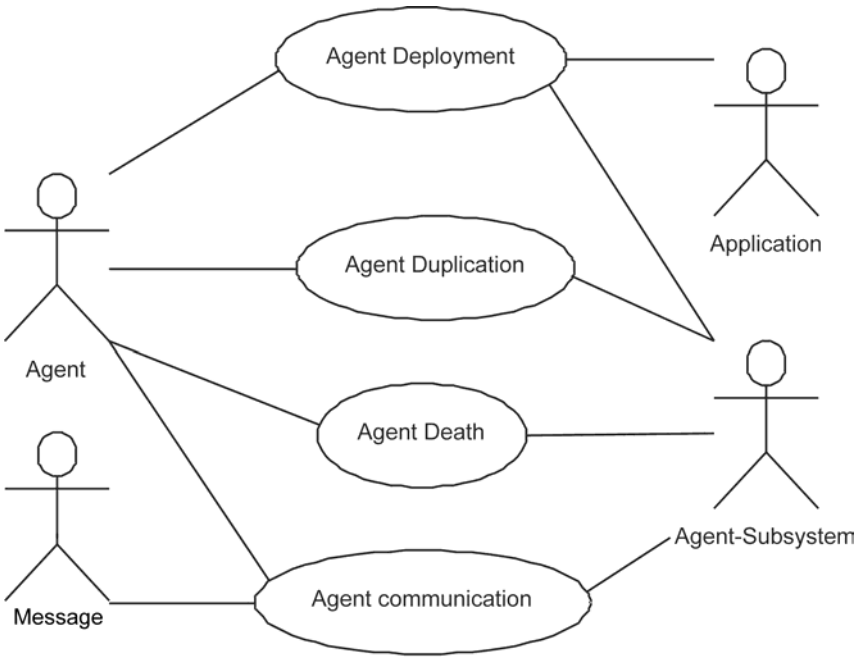

Figure 5-1: Agent-related use-cases

Agent-related use-cases are use-cases which are dealing with agents' actions and their life-cycle. These use-cases are illustrated in figure 5-1.

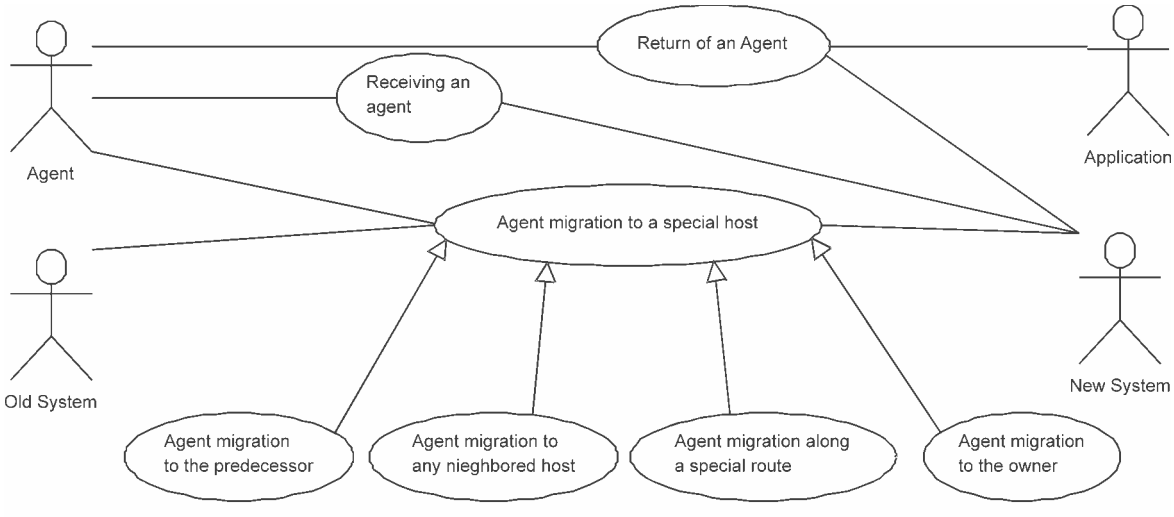

Figure 5-2: Migration-related use-cases

Migration-related uses-cases are dealing with the agents' migrations, i.e. moving to other systems. These use-cases are shown in figure 5-2.

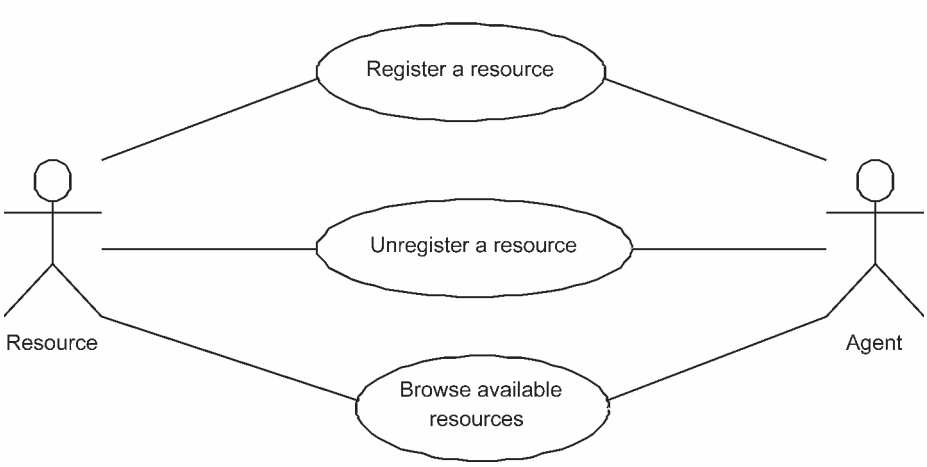

Figure 5-3: Resource-related use-cases

Resource-related use-cases are those which are dealing with the assignment and usage of resources. These are illustrated in figure 5-3.

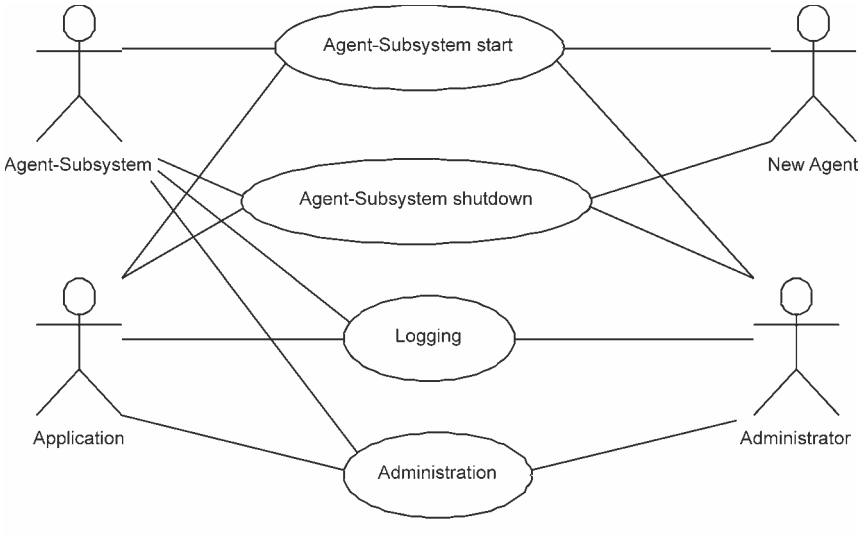

Figure 5-4: Agent-subsystem related use-cases

Finally, agent-subsystem related use-cases are those which are dealing with basic system functionality and services. They are shown in figure 5-4.

Within this chapter the functionality which will later be provided or supported by the framework will be called the “agent-subsystem” of the application which is also an actor for the use-cases. Later during the design phase this subsystem will be decomposed into different functionality domains. The use-cases are given in a tabular form described by Oestereich (Oestereich (2001)).

This framework, like any other framework, is supposed to support a special kind of application. It shall support applications relying on mobile agents in possibly unsafe peer-to-peer networks. The framework will not only provide a design to the application developer but also a standard or basic implementation of often used functions and integral parts for the management of agents.

However, it is also important to mention, which functionality will not be provided by the framework: The framework will only provide interfaces to peer-to-peer networks, but will not provide a peer-to-peer implementation of any kind. The reason for this is that there are many different and incompatible protocols available, so adding another one would make no sense. However, there are many implementations available, e.g. use the JXTA (see JXTA (2003)) framework developed by Sun Microsystems™ can be used.

5.1.1 Use Case: Agent-Subsystem start

Actors:	Application, agent-subsystem, new agent
Pre-Conditions:	- All providers are registered
Post-Conditions:	- Agent-subsystem is started - Agent-subsystem is connected to the peer-to-peer-network - Agents have been restored and are running again

Tab. 5-1 Properties of the use-case “Agent-Subsystem start”

This use-case, as outlined in table 5-1 describes the agent-subsystem start initiated by the application. Before, the application has to register all necessary functionality, like the peer-to-peer network implementation or the initial security policies. The agent-subsystem will then connect to the network and initializes the basic security measures as well as internal structures to manage the agents.

5.1.2 Use Case: Agent-Subsystem shutdown

Actors:	Application, agent-subsystem, agent
Pre-Conditions:	- Agent-subsystem is running
Post-Conditions:	- All agents' states are saved - All agents are stopped - Agent-subsystem disconnected from peer-to-peer network

Tab. 5-2 Properties of the use-case “Agent-Subsystem shutdown”

If the agent-subsystem is stopped by the application, it has to take care of the currently executed agents. It sends a message informing the agents of the shutdown so that these are able to relocate to other machines or kill themselves.

If an agent does not want to do this, it is saved before the shutdown so it can be restored when the agent-subsystem is started again.

The properties of this use-case are described in table 5-2.

If there were any agents running during the last shutdown these should be restored and restarted again.

5.1.3 Use Case: Administration

Actors:	Application, agent-subsystem, agent
Pre-Conditions:	- Agent-subsystem is running
Post-Conditions:	- Required action is performed

Tab. 5-3 Properties of the use-case “Administration”

Administrative instances like monitoring tools or administrators themselves should be able to kill an agent, stop or restart its execution and force the agent to move away from the current system. These functions are available in the agent-subsystem from the former use-cases but must be available and thus exported to the whole application utilizing this framework.

The properties of this use-case are described in table 5-3.

5.1.4 Use Case: Logging

Actors:	Application,.agent-subsystem
Pre-Conditions:	- Agent-subsystem is running
Post-Conditions:	- Application is notified of the occurring of loggable events

Tab. 5-4 Properties of the use-case “Logging”

Logging is a very important function for complex systems: It makes debugging and monitoring systems possible and also a lot easier. Also, it is necessary for performance analysis. The agent-subsystem should provide logging functions to the application which contains the message passing in the system, agents' starts and stops and resource access.

The properties of this use-case are described in table 5-4.

5.1.5 Use Case: Agent Deployment

Actors:	Application, agent-subsystem, agent
Pre-Conditions:	- Agent-subsystem has been started
Post-Conditions:	- New agent has ID - Responsibility for new agent has been transferred to the agent-subsystem - New agent has security policy - New agent is running

Tab. 5-5 Properties of the use-case “Agent Deployment”

An agent deployment means that a new agent is created and deployed to the local node. The properties of this use case are described in table 5-5. The agent has to be created by the application and deployed to the agent-subsystem. It is necessary that the application does not only provide an object, but the code to this agent as well, because it needs to be transferred over the network. The subsystem will create a new identification for that agent which is returned to the application and will choose an appropriate security policy. The identification can be used to identify uniquely the agent within the whole network.

Furthermore responsibility for the agent will be transferred to the agent-subsystem which from then on will manage the agent and start it as soon as possible.

5.1.6 Use Case: Agent Duplication

Actors:	Agent-subsystem, agent
Pre-Conditions:	- Agent is running
Post-Conditions:	- Agent is duplicated - New agent has security policy - New agent is running

Tab. 5-6 Properties of the use-case “Agent Duplication”

For some applications it is useful to duplicate an agent, for example if someone wants to traverse a network and wants to do this with more than only one agent. The properties of this use-case are described in table 5-6.

The agent calls the agent-subsystem to duplicate it. The new agent will then be created and initialized by the agent-subsystem like any other new agent introduced by the application.

The new agent will not have any resources registered.

Note that it is only possible to duplicate an agent within one container. If the application's need is to have this new agent to run at another node it has to ensure that the new agent will initiate a migration to the node afterwards.

5.1.7 Use Case: Agent migration to any neighboured host

Actors:	Agent-subsystem, agent
Pre-Conditions:	- Agent running
Post-Conditions:	- Agent is running on new node - Any resources are invalidated on the old host

Tab. 5-7 Properties of the use-case “Agent migration to any neighboured host”

The agent requests itself to be migrated to any other host. This might be useful before the actual node is shut down or the agent wants to search for resources randomly.

The agent notifies the agent-subsystem that it wants to be transferred to a neighbouring host. The container will stop the agent, assemble the code and the current agent's state and orders the peer-to-peer network to transfer this package to any known hosts. The properties of this use-case are described in table 5-7.

5.1.8 Use Case: Agent migration to a special host

Actors:	Agent-subsystem, agent
Pre-Conditions:	- Agent running
Post-Conditions:	- Agent is running on new node - Any resources are invalidated

Tab. 5-8 Properties of the use-case “Agent migration to a special host”

The agent requests itself to be transferred to another, known host. This might be useful to travel to a special node known to provide useful resources. The properties of this use-case are described in table 5-8.

The agent notifies the agent-subsystem that it wants to be transferred to that host. The container will. stop the agent, assemble the code and the current agent's state. Afterwards it runs a search within the peer-to-peer network for the given host. If that host is

or becomes available it orders the peer-to-peer network to transfer the package containing the agent and its state to that host.

5.1.9 Use Case: Agent migration along a special route

Actors:	Agent-subsystem, agent
Pre-Conditions:	- Agent running
Post-Conditions:	- Agent is running on new node - Any resources are invalidated on the old host

Tab. 5-9 Properties of the use-case “Agent migration along a special route”

An agent wants to travel to a known node within the peer-to-peer network. This is useful to gather resources known to be distributed along different machines. The properties of this use-case are described in table 5-9.

The agent notifies the container that it wants to visit these nodes and the container will assemble a package as in the “agent migration to a special host” but will add routing information for the path. It then proceeds as if the agent is to be transferred to one other host only.

If the current node is not the origin of the agent, it uses the saved path information to send the package to the next node in the path.

5.1.10 Use Case: Agent migration to the predecessor

Actors:	Agent-subsystem, agent
Pre-Conditions:	- Agent running
Post-Conditions:	- Agent is running on new node - Any resources are invalidated

Tab. 5-10 Properties of the use-case “Agent migration to the predecessor”

An agent wants to return to its owner travelling along the same path it used to travel to the current node. The properties of this use-case are described in table 5-10.

The agent notifies the current execution environment which stops the agents like in the use-case “agent migration to a special host” where the special host is the system from where the agent was received.

5.1.11 Use Case: Agent migration to the owner

Actors:	Agent-subsystem, agent
Pre-Conditions:	- Agent running
Post-Conditions:	- Agent is running on owner-node - Any resources are invalidated

Tab. 5-11 Properties of the use-case “Agent migration to the owner”

An agent wants to return immediately to its owner. It notifies the agent-subsystem where it is currently being executed which will extract the owner node of the agent from its meta-information and then will continue as in the use-case “agent migration to a special host”.

The properties of this use-case are described in table 5-11.

5.1.12 Use Case: Receiving an agent from the peer-to-peer network

Actors:	Agent-subsystem, agent
Pre-Conditions:	- Agent-subsystem is connected to the peer-to-peer network
Post-Conditions:	- Agent's former node is saved - Agent has security policy - Agent is running on node

Tab. 5-12 Properties of the use-case “Receiving an agent from the p2p-network”

A request from a neighbouring node is processed by the peer-to-peer network system which transfers the request to the agent-subsystem. There the agent's code as well as an image of its internal state is unpacked and any further information, like routing information is extracted and saved locally. Furthermore the agent-subsystem saves the node from which the agent came.

The agent is then restored, including its state and the setup of the security settings which apply to it. The agent is now ready to be started in a new thread. Note that resources the agent wants to use need to be re-registered after the transfer.

The properties of this use-case are described in table 5-12.

5.1.13 Use Case: Return of an agent

Actors:	Application, Agent-subsystem, agent
Pre-Conditions:	- Agent is running on owner node
Post-Conditions:	- Agent is stopped - Agent-reference is passed to the application

Tab. 5-13 Properties of the use-case “Return of an agent”

After gathering data an agent needs to deliver that data. If it has been migrated to its owner machine it can tell the agent-subsystem that it has completed its job. The agent-subsystem will stop the agent and inform the application that this agent has successfully returned. The agent-subsystem will provide a reference to the stopped so that the application can extract the gathered data.

The properties of this use-case are outlined in table 5-13.

5.1.14 Use Case: Register a resource

Actors:	Agent-subsystem, agent, resource
Pre-Conditions:	- Agent running
Post-Conditions:	- Resource is allocated - Agent receives resource

Tab. 5-14 Properties of the use-case “Register a resource”

An agent normally needs supporting resources where it can use data from in order to accomplish its task. The properties of this use-case are described in table 5-14.

Resources might be physically available like files and databases or pure virtual resources like shared memory.

Therefore, the agent can request a resource identified by the resource type, a colon and the resource's name, e.g. file:names.txt. If the resource is available, the agent-subsystem will add this resource to the agent's available resource-list. The agent is now free to use this resource.

Note that other systems abstract all resources as agents and allow the resources to be controlled via the grant of message-passing-rights as in the use-case “agent communication”. However, local resources are treated differently in this framework, because:

- File access and other operations are normally performed synchronously in applications because they are easier to handle than asynchronous requests which are sending several messages to the calling agent. These messages need to be processed in other event handlers, thus destroying the control flow of the methods. Note that message passing to another agent in this framework always is an asynchronous operation because the other agent is executed in a different thread.

- In general, message passing is possible between agents running on two different nodes within the network. However, for efficiency and security the use of resources should be limited to the local machine.

- Message passing instead of directly accessing a resource adds overhead and a corresponding, unnecessary penalty in execution speed.

5.1.15 Use Case: Browse available resources

Actors:	Agent-subsystem, agent
Pre-Conditions:	- Agent running
Post-Conditions:	- Agent receives virtual resource containing all available resources

Tab. 5-15 Properties of the use-case “Browse available resources”

An agent might not be able to know what resources are available on the current system so it is necessary that the agent can obtain a list with all resources. This list is always a resource which exists on every node called “resources:”. If the resource's type is known an agent might obtain a list from “resources:file” which will only list all files available.

The resource-list is a normal but pure virtual resource which contains a list of valid resource-names the agent can register and handle like any other resource.

The properties of this use-case are described in table 5-15.

5.1.16 Use Case: Unregister resources

Actors:	Agent-subsystem, agent
Pre-Conditions:	- Agent running - Agent has registered at least one resource
Post-Conditions:	- Resource is invalidated

Tab. 5-16 Properties of the use-case “Unregister resources”

After reading and perhaps writing data to resources, the agent can unregister itself from the resource, saving resources of the host system. This should always be done even though the agent-subsystem takes care of the unregistering when the agent is not executed anymore on the system.

The properties of this use-case are described in table 5-16.

5.1.17 Use Case: Agent communication

Actors:	Agent-subsystem, agent, message
Pre-Conditions:	- Agent running
Post-Conditions:	- Message is sent to its destination

Tab. 5-17 Properties of the use-case “Agent communication”

Agents are parts of software which can collaborate, so a mechanism is required which allows the communication from agent to agent. Furthermore it might be useful that the whole application might send messages to a specific agent, like commands received from the user interface. The properties of this use-case are described in table 5-13.

These messages contain the sender, the receiver and the message body, containing commands and/or data. The sender will pass the message to the agent-subsystem which will forward the message either to a local agent or send the message over the peer-to-peer network to another node.

The commands may be encoded depending on the application's needs, however there is a standard called Agent Control Language (ACL) for which implementations do exist so that it should be a practical choice.

If the agent wants to send a message to the local part of the application it can use the reserved name “host” as the message's destination.

The properties of this use-case are described in table 5-17.

5.1.18 Use Case: Agent death

Actors:	Agent-subsystem, agent
Pre-Conditions:	- Agent is running
Post-Conditions:	- Agent is killed - All resources all invalidated

Tab. 5-18 Properties of the use-case "Agent death"

If the agent realized it has failed or it has accomplished a job without needing to deliver data it can destroy itself by telling the agent-subsystem to kill it which will stop the agent, unregister all resources and delete all information associated with the agent.

The properties of this use-case are described in table 5-18.

5.2 Security

Mobile agents raise lots of security issues because they are implemented using mobile code which is normally untrusted, especially if it is in an anonymous peer-to-peer network.

To allow the safe execution the framework has to provide a rich set of security measures, like asymmetric encryption and signing[19] of code and data. Furthermore the framework should limit the actions an agent may execute, e.g. do not make all files accessible.

Agents should be anonymous, that means the owner is unknown, or should prove the owner with a passport. This passport is a ticket with the agent's name, its maximum lifetime and a signature of the owner as described in the OpenPGP Transport Standard.

The agent's code can be signed by the programmer as well, so that the agent's developer is identifiable.

The framework should provide so called security policies which can be associated with an agent and control the rights it has on the current node. The security policy can control whether an agent may access resources, write to them, pass messages etc. Whenever

[19] Asymetric encryption and digital signatures are described in the chapter OpenPGP Transport Security

an agent is executed on the system a matching security policy has to be selected within the framework. This selection can be based on the developer, the owner and the name.

For securing data in transfer, encryption and signing of the agent and its state is necessary. To secure a path an agent wants to travel, an easy way is outlined in “Securing Your Data in Agent-Based P2P Systems” (Pang (2003)).

During the development of the framework security especially means that input has to be validated before it can be trusted and no references to generic objects like System.Object in Java™ may be passed to agents. The latter ensures that no resources bound to an agent are transferred to another.

The same problem exists, if an object with a specific interface is passed to an agent, but the object has more publicly accessible methods. It should be guaranteed that the passed object only provides public methods specified in this interface. If this is not the case, proxy classes[20] have to be used. This is necessary because the agent is able to cast the object to its full interface and then accesses the “hidden” methods as shown in following example:

```
class MyImpl implements MyInterface
{
  public void methodNotInInterface() { ... }
}

class MyAgent extends Agent
{
  public void aMethod(MyInterface object)
  {
  MyImpl i = (MyImpl)object; // cast here
  i.methodNotInInterface(); // should not be possible!
  }
}
```

5.3 Flexibility

To be useful for as many application scenarios as possible, the framework should be very flexible in terms of customization. It should be possible to exchange algorithms or plug in new functionality. This should apply to user interfaces, security policies, message standards and peer-to-peer network implementations. However, this requirement does conflict with the framework's requirement of ease of use, because the more modules are exchangeable, the more methods are required for the registration process.

[20] For a description of the Proxy design pattern see Gamma (2001), pp207ff.

6 Framework Overview

6.1 General

The framework is implemented in Java™[21], a programming language and platform developed by Sun Microsystems™. Java™ has been chosen because of the benefits it offers in the given application domain:

- Portability
  Java™ programs run on top of a so called Java™ virtual machine, which creates a virtual platform. Programs are stored in Java™ byte-code which is transformed to platform specific machine code during runtime. This way a Java™ program can run on any computer, which has a Java™ virtual machine installed. Sun Microsystems™ offers Java™ virtual machines in the Java™ runtime environment (JRE) for several platforms free of charge, including Linux™, Windows™ and Solaris™. Other companies, like IBM™ are offering a JRE as well for their systems.

- Security
  Java™ has integrated security concepts since its first version: This way it is possible to ensure that certain code may not access local files etc. This technique has been used for Java™ Applets[22], small programs which are embedded into web pages and executed within the browser but may not access any local resource and only establish connections to server, from which they are loaded. The restriction of Java™ Applets is also called sandboxing and can be applied to any code, including mobile agents. Furthermore, Java™ only uses object references and has no pointers, making direct memory access impossible. This way, buffer overflows, where bugs in applications are used to inject code, are impossible. The same is valid for the problem of dangling pointers: A reference always points to a valid object or has the value null. So it is not possible to access already deleted objects.

- Productivity
  The use of references instead of pointers not only supports the security of an application, but also enhances the programmer's productivity because programmers cannot directly access memory and therefore make fewer mistakes. Furthermore, the destruction of objects is handled by the JVM: A garbage-collector process is running in the background and freeing objects which are not referenced anymore, which improves the development time and reduces the number of bugs.

[21] Further information about Java™ can be found under http://java.sun.com
[22] Further information about Java™ applets can be found unter http://java.sun.com/applets/

- Tools
  Not only the Java™ language itself is very convenient, but there are also many comfortable development environments available, like NetBeans by Sun Microsystems™ (Sun Microsystems (2003b), Eclipse initiated by IBM™ (Eclipse (2003)) and JBuilder by Borland (Borland (2003)). These tools are designed for supporting large projects integrating refactoring facilities, project management and further functionality, like Unified Modelling Language[23] (UML) support.

For developing the framework, Eclipse has been used, because it has a very good editor and refactoring support for editing class structures during the whole developing process.

6.2 Package structure

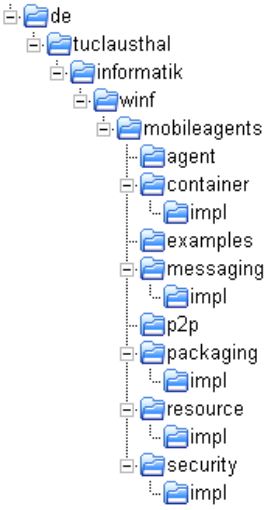

Figure 6-1: Package structure of the framework

The framework is decomposed into seven essential packages and one example package, in which all example code is placed. The hierarchy is illustrated in 6-1. The root package is, conforming to Sun Microsystems'™ standard (see Sun Microsystems (1999)):

```
de.tuclausthal.informatik.winf.mobileagents
```

Because this name is very long, all names are abbreviated as following:

`.agent.Agent` is `de.tuclausthal.informatik.winf.mobileagents.agent.Agent`.

The seven essential sub-packages each represent one functionality domain of p2p-applications based on mobile agents. The packages mostly contains interfaces which

[23] For an introduction to UML see Oesterreich (2001)

have to be implemented by the application.. The package structure should also be used for own developments, that means, that if an agent is implemented, which means you are building a class which implements the interface `.agent.Agent`, it should placed in a package `yourapp.agent`. The seven sub-packages, containing the design for their specific functionality domain, are structured as following:

- .agent
  This package contains all interfaces and abstract classes needed to implement an agent, especially the interface `Agent` which has to be implemented by all agents.

- .container
  This package contains all interfaces which are needed for an own container-implementation. A `Container` represents the execution environment for the mobile agents, which has to manage agents, their messages and resources.

- .messaging
  The messaging-package contains all interfaces and abstract classes needed for providing messaging services and message management.

- .p2p
  The p2p-package primarily contains the `P2PNetwork` interface, which provides the methods for accessing a p2p-network. Use this package to directly access a p2p-network or, more importantly to implement a p2p-network protocol.

- .packaging
  This package contains the interfaces needed for packaging an agent, that means storing and restoring it so that it can be transferred over the network and be resume execution on the other node.

- .resource
  The resource-package contains the interfaces for dealing with the resources and their management. This framework deploys its own abstraction layer for accessing all kind of resources, which are accessible by the agents, like files, databases etc. The resource-management interfaces and the interface `Resource` are located in this package.

- .security
  This package contains interfaces for the use of cryptographic code in this framework. Furthermore, the permission-handling encapsulated in the interface `SecurityPolicy` is included here. Besides the `.container` package, this is the place, where the sensitive code is located and consistently should be tested thoroughly.

It should not be neccessary to instantiate a class out of any package directly; instead there are managers available, which are singletons and managing their corresponding object spaces. All requests for objects should be directed to these managers to make full use of the loose coupling of the framework for better maintenance and maintainability.

Under most sub-packages there exists a further package `impl`. In these packages the basic implementations are located, like `BasicContainer`, which will be registered as default in the managers. Note, that there is no default implementation of a p2p-network because it is out of scope of this framework to provide a fully-fledged network and many p2p-networking protocols have been developed elsewhere. If an implementation is needed to experiment with, one can use the p2p-network from the sample application.

6.3 Packages

6.3.1 Package de.tuclausthal.informatik.winf.mobileagents

Figure 6-2: Class diagram of the managers in the framework

The base package of the framework contains so called manager-classes, as can be seen in figure 6-2. Managers are responsible for managing the functionality of a specific ap-

plication domain. All managers are implemented using the Singleton design pattern[24]. This pattern is used to guarantee that only one object of a specific class is instantiated ever during the lifetime of an application. To achieve this, the constructor for the class is declared private or protected, so that it cannot be accessed outside of objects belonging to the class. This results in a class which cannot be instantiated by any object belonging to another class in the application. However, to use the functionality provided by the singleton class, it has to be instantiated. For this, a static method, typically called `getInstance` is used which accesses a private or protected attribute, typically called `instance`. On the first call to `getInstance` the first and only object of the singleton is created and saved to the static member instance so that it can be accessed again. For example the `ContainerManager` is implemented this way:

```
public class ContainerManager
{
     [...]
     protected static ContainerManager instance = null;
     [...]
     public static ContainerManager getInstance()
     {
          if(instance==null)
          {
               instance = new ContainerManager();
          }

          return instance;
     }
}
```

The advantage of using managers is the very high degree of independence between the subsystems of the framework: Since all calls for acquiring specific instances of classes are directed to the corresponding managers, these are responsible for returning the right or currently to-use implementation. Because all return values are only references to interfaces and not to specific implementations, it is possible to change certain implementations without needing to update any other part of the application. For example, to get a `CryptographicProvider` object supporting the OpenPGP encryption standard, any application part has to call

```
CryptographicProvider cp = CryptographicManager.
          getInstance().getCryptographicProvider("OpenPGP");
```

The returned implementation can be whatever the current system uses, for example an implementation using GnuPG (like the default provider does) or a front end to PGP or some self-made implementation. This way the whole system is loosely coupled and very flexible.

[24] For more information about the Singleton design pattern see see Gamma (2001), pp. 127ff.

Following managers are defined by the framework:

- ContainerManager
  The `ContainerManager` is responsible for returning and managing the reference to the `Container` in charge. It is simply a singleton wrapper for the `Container` interface, allowing the central access to the `Container` in use by using the `getContainer` method.

- CryptographicManager
  The `CryptographicManager` is responsible for taking care of all `CryptographicProvider` in the application. Each `CryptographicProvider`, which offers cryptographic algorithms to the application, has to be registered with this manager using the `registerCryptographicManager` method. Afterwards all parts of the application may obtain a certain `CryptographicProvider` object by calling the `getCryptographicProvider` method.

- MessagingManager
  The `MessagingManager` is responsible for managing all of the `MessagingProvider` in use by the application. A `MessagingProvider` implements a certain protocol or virtual transport path for sending messages. Furthermore, the `MessagingManager` routes messages between these providers and thus is responsible for the message-sending and -routing. The third function fulfilled by the `MessagingManager` is its use as a factory for the `Message` objects in use by this application, which are used to represent and validate messages in the application.

- P2PManager
  The `P2PManager` is responsible for managing all p2p-networking-protocols in use by the application. Each protocol has to be registered via the method `registerP2PNetwork`. The `P2PManager` is responsible for managing these and can be queried for specific protocols via its method `getP2PNetwork`. The `P2PManager` therefore is the central access point for everything p2p-related.

- PackagingManager
  The `PackagingManager` is responsible for managing all instances of the interface `PackagingProvider`, which can be used to transform `Agent` objects into a binary representation suitable for sending it over the network. If some part of the application needs this kind of functionality it has to acquire the corresponding provider via these managers.

- ResourceManager
  The `ResourceManager` is used to manage the resource-providers of the system. A `ResourceProvider` offers local physical and virtual resources to the agents. If an agent requests an `Resource` object the request has to be routed to the `ResourceManager`, which will find the corresponding `ResourceProvider`. The `ResourceProvider` will then create a `Resource` object which can be used for reading and writing data.

- SecurityPolicyManager
  The `SecurityPolicyManager` is responsible for assigning a `SecurityPolicy` to an `Agent` object. All incoming agents need to have a policy assigned, which will define their access rights on the local system. For example, every `Packager`, which reconstructs a transmitted agent, has to use the `SecurityPolicyManager` to let a policy be assigned to the agent.

- ServiceManager
  Services are extensions of the system which allow agents the use of further operations and access to the system, which are not predefined in the framework. These services have to be registered with the `ServiceManager` so they are accessible for the system and can be requested on the agents' behalf.

The managers are integral to the framework: They define the control flow and are used frequently to obtain implementations of specific functionality domains. This way they shield the different subsystems from each other and allow easy interchangability of implementations.

6.3.2 Package .agent

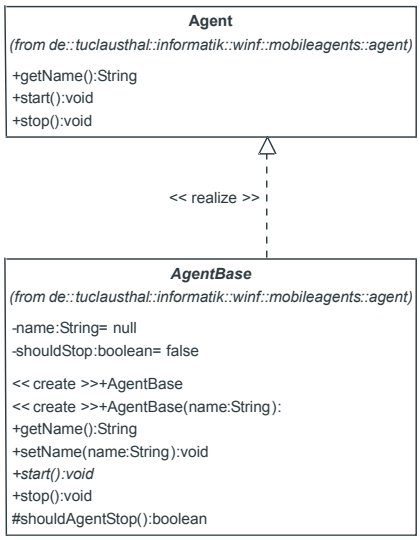

Figure 6-3: Class diagram of the package agent

The sub-package `agent`, as illustrated in figure 6-3, contains the interface `Agent` and an abstract base class `AgentBase`. An agent has to implement the interface `Agent`. Its main logic is placed inside the method `start`. This method is called whenever an agent should start execution. The method takes a reference to an `AgentServices` object, which the agent must use in order to access resources and functionality of the host system.

An agent is notified by the container that it should exit its `start`-method with a call to the method `terminate`. If the agent does not exit afterwards the system is allowed to kill the agent's thread.

The interface `Agent` extends the `Serializable` interface, which can be used for transforming an object into a byte-stream. The implementation is done transparently by the compiler, reducing implementation time.

The abstract base class `AgentBase` provides a rudimentary implementation of all methods except the logic which has to be implemented in a derived class in the `start`-method.

A simple agent, as included as `EchoAgent` in the examples, may look like this (most comments have been stripped out for clarity and conciseness):

```
public class EchoAgent extends AgentBase
{
     public void start(AgentServices agentServices)
     {
          while(!this.shouldAgentStop())
          {
```

```
                // wait for a message
                // method clears message so we don't need to delete it
                Message m = agentServices.waitForNextMessage();

                // m can be null, e.g. if we should quit
                if(m == null) continue;

                if(m.getBody().equals("EXIT"))
                {
                    return;
                } else
                {
                    // create a new message, agentServices work as a fac-
        tory for us
                    Message echo = agentServices.createMessage();
                    // set the recipient and the body
                    echo.setRecipient(m.getRecipient());
                    echo.setBody(m.getBody());
                    // we do not want a receipt
                    echo.setRequiresReceipt(false);
                    // send message
                    agentServices.sendMessage(echo);
                }
            }
        }
}
```

This example demonstrates how easy it is to implement an agent. The only thing one should pay close attention to is the termination of the main-loop, so that the agent exits safely and predictably. Any pending messages and open resources will be closed and deleted by the execution environment. If an agent should stop execution can be queried by using the method `shouldAgentStop`. This method is provided by the `AgentBase` abstract class for convenience. If an agent is written from scratch using the `Agent` interface, the programmer has to check, whether the agent's thread has been interrupted by using the method `Thread.isInterrupted` or if the agent's `stop`-method has been called.

6.3.3 Package .container

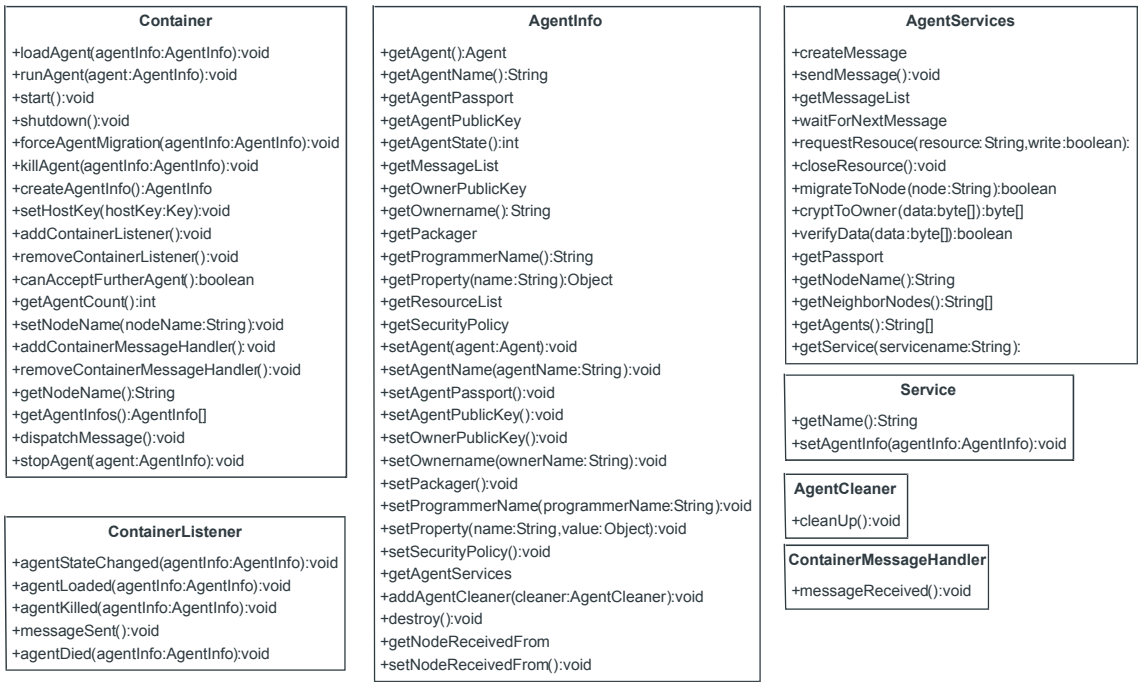

Figure 6-4: Class diagram of the package container

The `.container`-package, as shown in figure 6-4, contains the interfaces for building so called `Containers`, which are execution environments for the agents. Their task is to manage all agents and their related resources. Furthermore, a `Container` object is responsible for serving the agents' requests, like message passing. After all, agents' requests are submitted to an interface `AgentServices` which has to be implemented as part of the container-implementation. The `AgentServices` provide the agent with all the options it has. These include

- message passing,
- migration to other hosts,
- obtaining neighbouring nodes in the network and
- cryptographic services.

If a platform wants to provide more services, it has to do so by implementing the functionality into a class which implements the `Service` interface. Each `Service` can then be registered with the `ServiceManager` and acquired by an agent over its `AgentServices` instance.

For security reasons the `AgentServices` should only be implemented in an extra class and not as a part of the main container, so that type-casting attacks are not possible. For this reason, the methods are strictly separated into two interfaces.

There exists an interface `AgentInfo` for managing agents. A `Container` should manage all agents through objects implementing this interface and store the appropriate properties therein. For most things there are standard accessor methods (`getX` and `setX`) available. For non-standard information, the `AgentInfo` provides a `getProperty` and `setProperty`-method, which can retrieve and store named objects. Note that other parts of the application will likely access the `AgentInfo`, for instance a `Packager` object, which is responsible for transforming the `Agent` from a network byte-stream into an executable form, and will store further management-information to the `AgentInfo`. However, it is up to the `Container` how the `AgentInfo` should work. For example, it is possible to map an `AgentInfo` to another internal managing object, making it a proxy. Because of this, the container acts as a factory for its `AgentInfo` object: it has a method `createAgentInfo` which returns an AgentInfo. This can be picked up and filled with information by a `Packager`.

A `Container` object can be observed by a corresponding `ContainerListener` as specified in the observer pattern[25].

Events triggered to a `ContainerListener` are:

- agents are created or killed,
- agents states are changing, and
- messages are sent.

To subscribe to events the `registerContainerListener`-method has been invoked. This registers the given object so that it will be notified of further events by calls to the methods defined in the listener's interface. The complete process is illustrated in figure 6-5.

[25] For more information about the Observer design pattern see Gamma (2001), pp. 293ff.

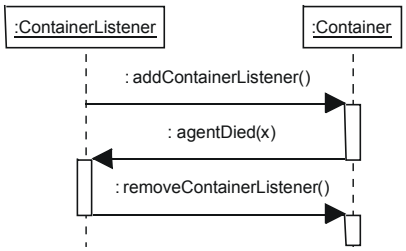

Figure 6-5: Sequence diagram for the ContainerListener interface

Furthermore, messages sent to the `Container`, e.g. the system itself, can be intercepted by objects adhering to the `ContainerMessageHandler` interface. This way, it is possible for remote agents to send messages to their home system or to log specific message traffic.

6.3.4 Package .messaging

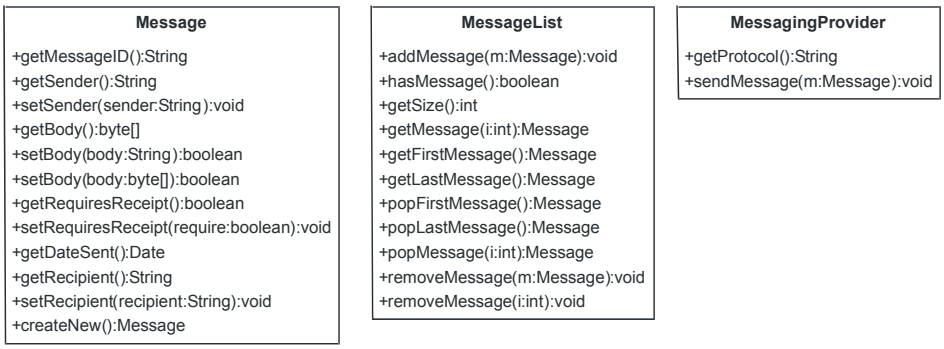

Figure 6-6: Class diagram of the package messaging

The `.messaging`-package contains the interfaces which are necessary to implement message-passing over the p2p-network as well as between local agents. The interfaces are illustrated in figure 6-6. The central interface is `Message`, which defines all attributes of a `Message`, which can be passed from one entity, like agent, peers etc., to another. Essentially a message is a byte-array – a so called body - which should be sent to a recipient and has a sender. For convenience, a `String` may be set as the body which is transformed to a byte-array.

The sender may require a receipt, which means that if the message is delivered, a confirmation is sent to the sender.

Recipients and senders are identified by their corresponding address. An address is defined as

`protocol:entity[@peer]`

and consists of three parts:

- the protocol, which identifies which protocol the system has to choose to deliver the message. For the peer, on which the message is send, simply use the local:-protocol,
- the entity, which can be an agent, a host or another virtual identity. An entity's name may contain all alphanumerical characters and the underscore (A-Z, a-z, 0-9 and _), and
- the destination peer, which is optional and appended with an @-sign at the entity-name. The peer describes to which peer within the network, the message should be routed to. The peer-name has to adhere to the rules which are valid for the specified protocol and has to be reachable by that protocol as well.

There is one special address defined in this framework, which is `local:host`. All messages sent to this address should be forwarded to the `Container` object in-use, which in turn will notify all its `ContainerMessageHandlers`. Additionally, all messages send via the local:-protocol are never allowed to leave the local node.

If the application wishes to change the address structure, it has to reimplement the corresponding `MessagingManager`, because there the parsing of addresses and the corresponding routing is done.

6.3.5 Package .p2p

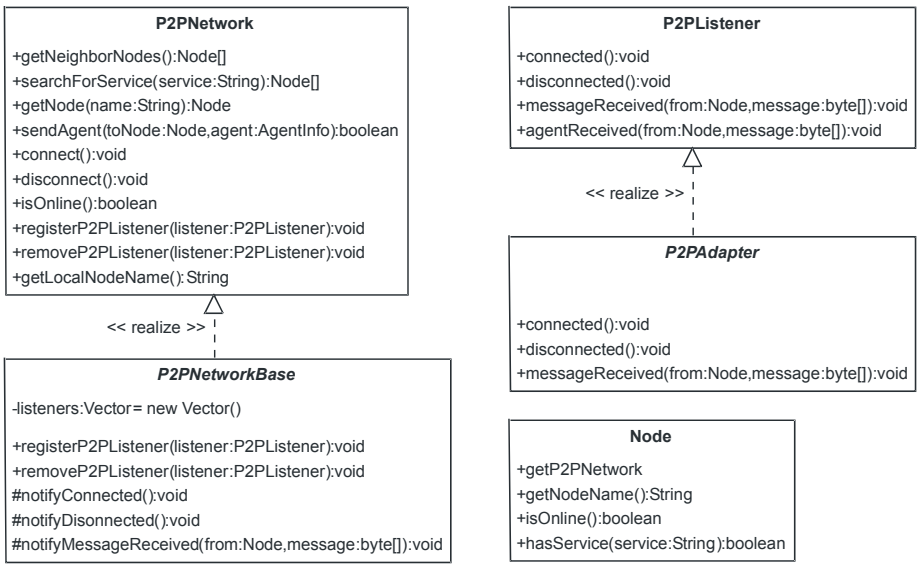

Figure 6-7: Class diagram of the package p2p

The purpose of the `.p2p`-package, as shown in figure 6-7, is to provide the interfaces for the peer-to-peer network component of the application. It mainly consists of the interface `P2PNetwork` and a corresponding listener interface. The listener may be used to monitor the network, e.g. when connections go up or down.

Note that this package does not enforce a specific p2p-network design: The network can be designed as one likes. This includes the internal classes for the implementation as well. Only the external interface has to match the interface `P2PNetwork`. For this reason, the `P2PNetwork` interface concentrates only of the framework's requirements, for instance, sending a message object or migrating an agent.

The p2p-network's nodes have to be accessible via the `Node` interface. Normally there will be a wrapper-class implementing this interface and encapsulating the official p2p-network address of a node and delegates all non-trivial method-calls to the `P2PNetwork` implementation. This way, the p2p-network implementation can determine the right return values and answer them accordingly. Furthermore, the `Node` implementation will be light-weight and possibly can be implemented as an inner class without degrading code readability.

6.3.6 Package .packaging

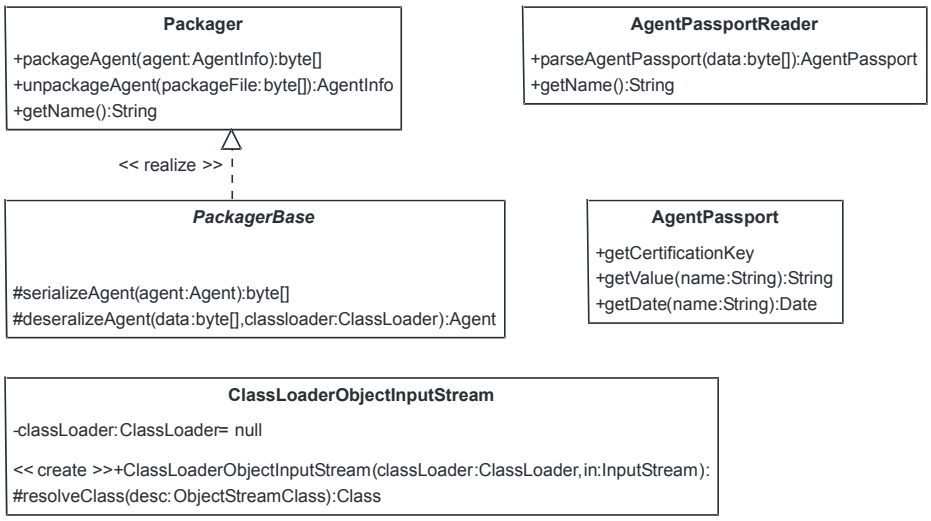

Figure 6-8: Class diagram of the package packaging

The `.packaging`-package, as illustrated in figure 6-8, is responsible for defining the interfaces which are used for serializing the agent to a byte stream and restore it when received. So called packagers as defined in the `Packager` interface are responsible for this. They are responsible for the whole process, which includes security checking and class-loader setup as well. This is necessary, because all the things are specific on how

an agent is packaged. However, the system should be built as modular as possible. This way the existing implementations are reuseable for other packagers and better maintainability is achieved. For example, the agent's passport, which is described in the OpenPGP transport security chapter, is read by an `AgentPassportReader`. One can utilize this `AgentPassportReader` for all kinds of packaging standards, as long as the format of the passport is identical.

Furthermore, for all cryptographic routines, one should use the corresponding `CryptographicProvider` which can be obtained by using the `CryptographicManager` (see the description of the security-package for more information).

Since most packagers will need to serialize the `Agent`'s data into a byte-array, there exists a `PackagerBase` abstract class, which introduces and implements two new methods: `serializeAgent` and `deserializeAgent`. They are provided for convenience to take the serializable `Agent`, read the `Agent`'s state and transform it into a byte-array. For this implementation the `ClassLoaderObjectInputStream` is used, which provides functionality for this process.

6.3.7 Package .resource

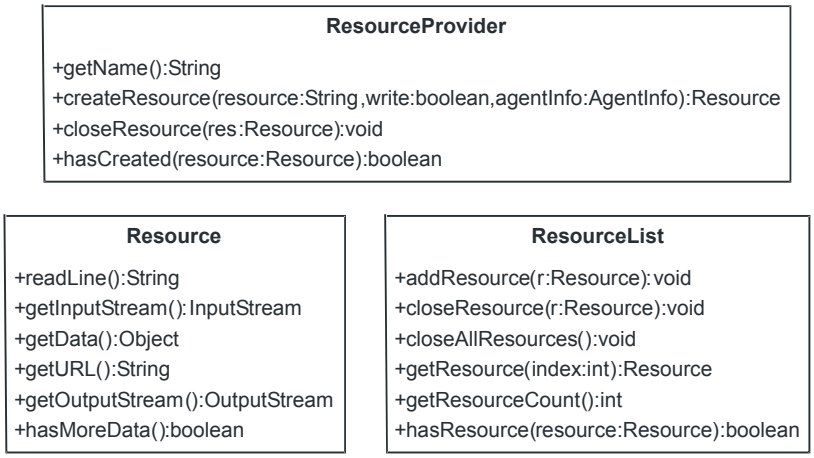

Figure 6-9: Class diagram of the package resource

In the `.resource`-package, all interfaces for managing the access of resources by agents are defined. This includes the `ResourceProvider` interface, which is responsible for creating and initializing `Resource` objects. A `ResourceProvider` has its own "protocol" like file:, mp3: or http: so that the agent simply accesses abstract resources in terms of a URL. By requesting a resource storage://mystorage the agent obtains a reference to a `Resource` object which it can use to read and/or write data to. The `Resource` interface

defines lots of access functions. This way, an agent may use streams to operate on the `Resource` or may read data line by line.

For managing `Resource` objects, a `ResourceList` is specified, which can be used by all implementations which need to keep track of their resources.

The class diagram of this package is provided in figure 6-9.

6.3.8 Package .security

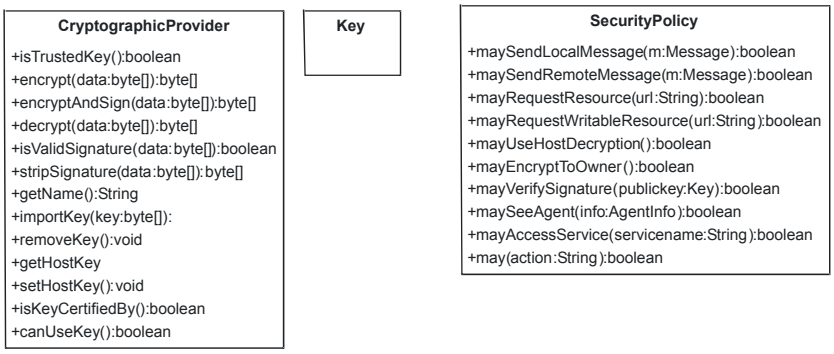

Figure 6-10: Class diagram of the package security

In the `.security`-package, as illustrated in the class diagram in figure 6-10, all security relevant interfaces are grouped together. On the one hand these are cryptographic related interfaces like the `CryptographicProvider` and `Key`; on the other hand there are access-control related interfaces like `SecurityPolicy`. The `.security`-package is therefore a critical component of the framework.

A `CryptographicProvider` serves as an interface for conducting all actions like signing, encrypting and decrypting data. The used keys are encapsulated in the interface `Key`.

The `SecurityPolicy` is used to store permissions an agent has on the local system. It is queried by the `AgentServices` for the agent before fulfilling a request.

6.4 Control Flow

6.4.1 Message Passing

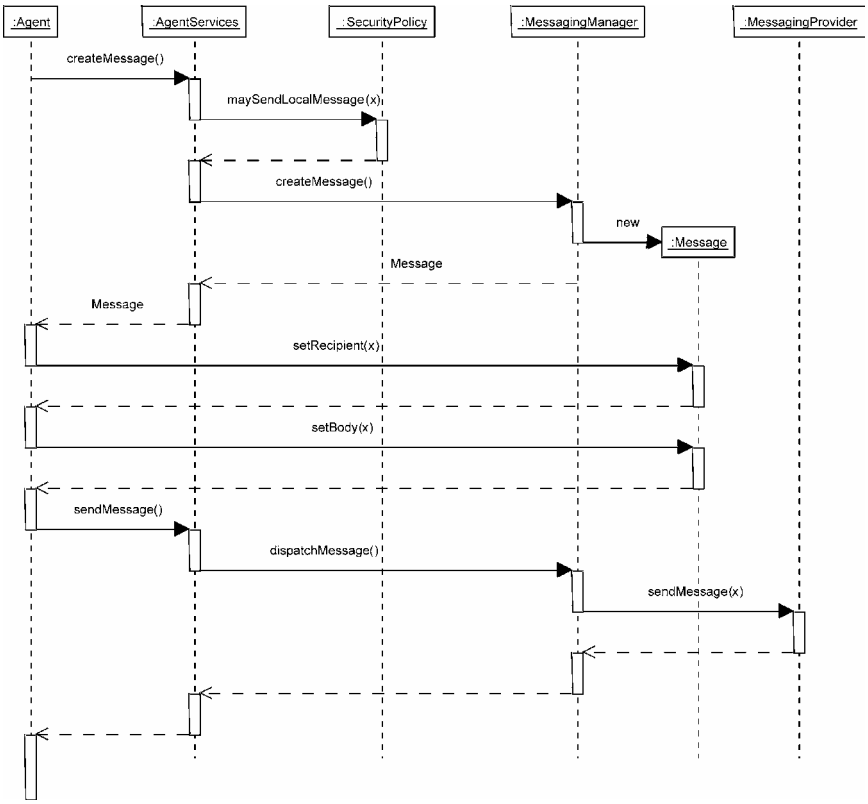

Figure 6-11: Sequence diagram for the message passing process

Whenever an agent wants to send a message, it has to create a new `Message` object by using its `AgentServices`' `createMessage`-method. The `AgentServices` normally will delegate the request to the `MessagingManager` which will act as a `Message`-factory as well. The agent may then set a recipient and the data to be transferred and submit the message via the `AgentServices` as well. The `AgentServices` will set the sender of the message to the agent's name and consult the agent's `SecurityPolicy`, if the agent is allowed to send the message. Whether the message's recipient starts with “local:”, the `SecurityPolicy` has to be asked if a local message or a remote message may be sent. Afterwards, the `Message` simply gets passed to the `MessagingManager`. The `MessagingManager` will pick the registered `MessagingProvider` for the recipients supported protocol and will submit the `Message` object to it. The `MessagingProvider` is then responsible for delivering the message.

The whole process is illustrated in the sequence diagram shown in figure 6-11.

6.4.2 Resource Requests

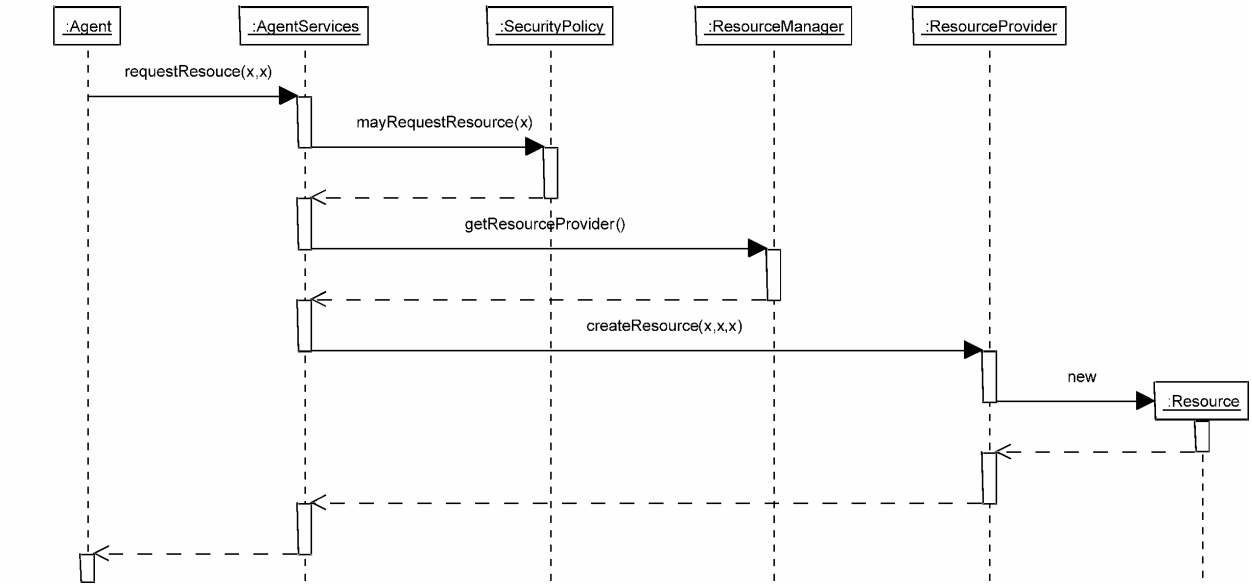

Figure 6-12: Sequence diagram for the resource request process

Requests for local resources have to be conducted via the `Agent`'s instance of `AgentServices` as well. The `AgentServices` have to check the agent's `SecurityPolicy` object, whether the resource-request is allowed or not. If it is, the corresponding `ResourceProvider` object can be obtained via one of the `ResourceManager`'s `getResourceProvider`-methods. The `ResourceProvider` can then be used to create a `Resource` object, which matches the request, by invoking its `createResource`-method.

6.4.3 Agent Migration

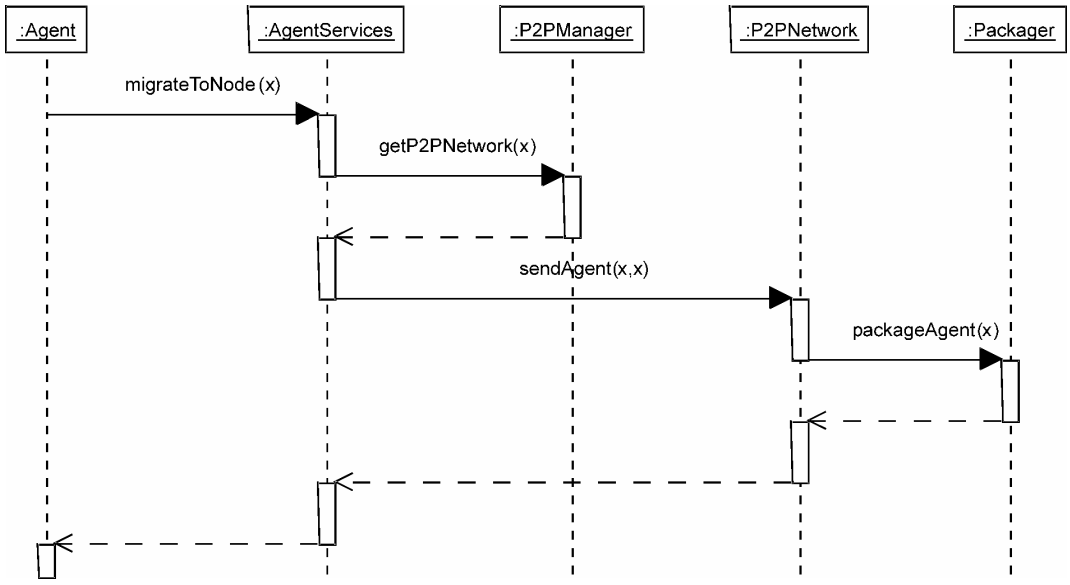

Figure 6-13: Sequence diagram for the agent migration process

A mobile agent can decide to travel to another host in the network. This is called migration. The agent therefore sends the migration-request to its `AgentServices` object. These are responsible for initiate the migration of the agent by delegate the migration-request to the p2p-network, which can be looked up via the `P2PManager`. The network implementation is responsible to choose the `Packager` for the migration. The agent's `AgentInfo` contains a reference to the `Packager` used to unpackage the `Agent` object and, if possible, should be used to package the agent again. The resulting byte-array can then be sent over the p2p-network, where the destination host is located. If the migration was successful, the `AgentServices` should return `true` to the `Agent`, otherwise they should indicate the failure via a `false` return value. The `Agent` object can then decide, if it ends its execution after the successful migration or if wants to go to another host as well etc.

The complete process is illustrated in a sequence diagram in figure 6-13.

6.4.4 Agent Acceptance

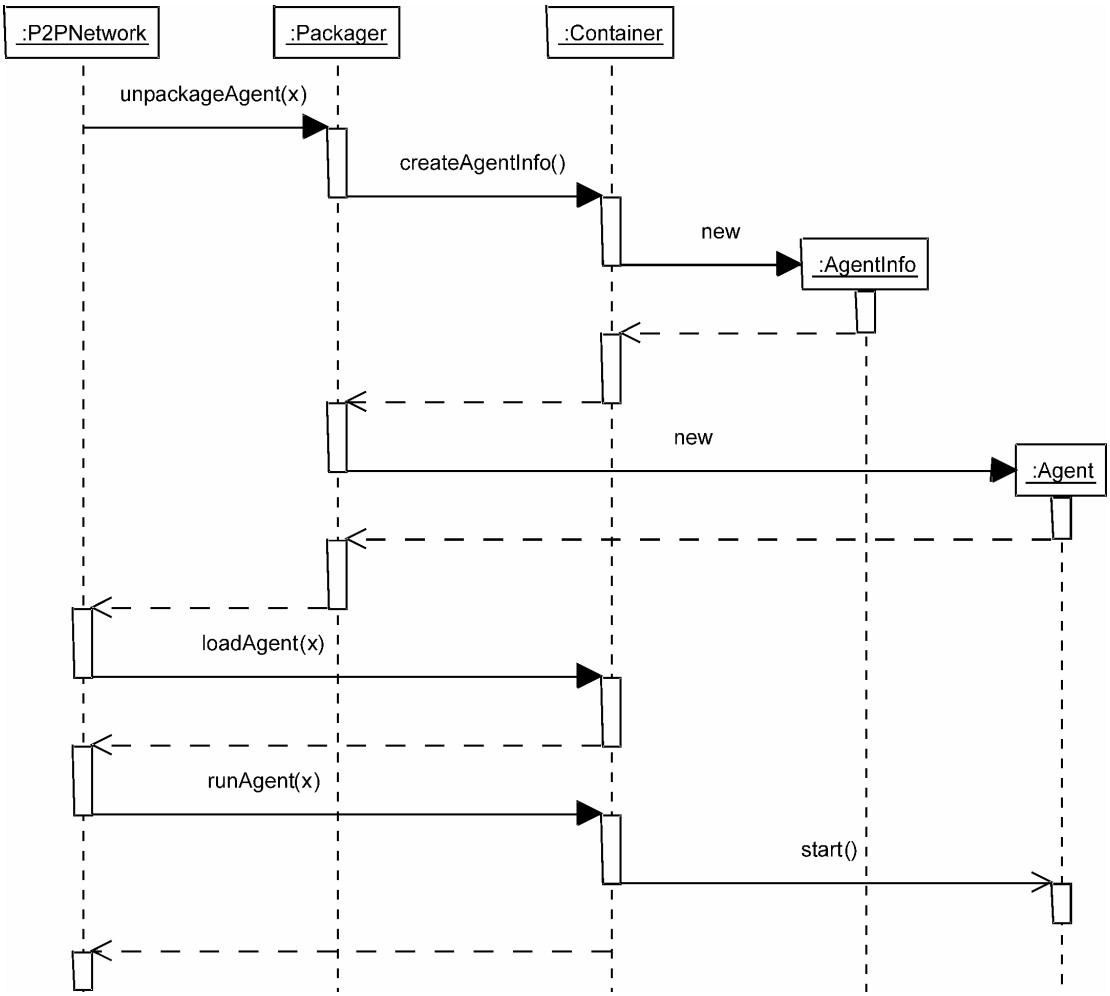

Figure 6-14: Sequence diagram for the agent acceptance process

If a `P2PNetwork` object receives an incoming request to accept an agent, it has to pass the binary representation to a `Packager` object. This `Packager` can be obtained by the `PackagingManager`. The `Packager` will transform the given byte-array into an `AgentInfo` object, which has a corresponding `SecurityPolicy`. The `P2PNetwork` has to set the reference to the `Node`, from which the agent came and can pass the `AgentInfo` object to the `Container` in use. The passing is done in two steps: First, the `AgentInfo` has to be loaded, which means all management information structures are updated and the `AgentInfo` will be checked. The second step is the start of the agent, where the agent's execution will begin.

The whole process is documented as a sequence diagram in figure 6-14.

6.5 Basic Implementation

6.5.1 .container.impl.BasicContainer

The `BasicContainer` provides a default execution environment for the system. It manages incoming agents and provides notification for all registered listeners. The implementation uses the `BasicMessageList`, `BasicAgentInfo` and `BasicAgentServices` classes to manage all agent-related information. For all agents a seperate thread is started which executes the agents' logic.

All requests issued by an agent via its `BasicAgentServices` instance are delegated to the `BasicContainer` or to the corresponding manager. For instance, the `createMessage` call is delegated to the `MessagingManager`. This way the `BasicContainer` can be used in very different scenarios if the required implementation is properly registered with their corresponding manager.

6.5.2 .messaging.impl.BasicMessage

The `BasicMessage` is a straight-forward implementation of the `Message` interface. It uses an internal byte-array to store the message's body and `String` objects for remembering the sender and the recipient of the message. It accepts anything as the message's body without restrictions to the size. If this presents a problem and one wants to limit the size of the message's body, simply override `setBody(byte[])`. This is sufficient, since `setBody(String)` will forward the request to the overloaded method. The same is valid for checking for specific body contents, e.g. checking if a message is ACL compliant or not: In either case `BasicMessage` can be subclassed and `setBody(byte[])` overriden.

6.5.3 .messaging.impl.BasicMessageList

A `BasicMessageList` manages incoming messages in a `Vector`, whose reference is stored in the instance variable `messages`. Essentially all operations are simply mapped to the `Vector`'s methods resulting in no method longer than three lines. This means that the `BasicMessageList` is simply an example for the adapter pattern to match the interface of the `BasicMessageList` to the `Vector`.

Note however, that the `BasicMessageList` does not have an upper limit for how many messages are stored, so that the list may consume too much memory, therefore being a target for denial-of-service attacks (DoS).

6.5.4 .messaging.impl.LoopbackProvider

The `LoopbackProvider` implements the local: protocol for message-passing. Every message which is addressed to a local recipient is routed to this provider and handed over to the `Container` in use. The `LoopbackProvider` will be registered automatically by the `MessagingManager`. If one wants to avoid this behaviour, one has to implement a new `MessagingManager`.

6.5.5 .security.impl.GPGCryptographicProvider

The `GPGCryptographicProvider` implements the OpenPGP standard for encryption and signing using the GNU Privacy Guard (GnuPG, see GNUPG (2003)), available under the General Public License[26] (GPL). It is available for lots of platforms, including but not limited to Linux, Windows and Solaris.

This implementation was done from scratch, because no free OpenPGP-compliant implementation in Java™ was found. The only implementation found was one for the Microsoft .NET platform (see Kartmann (2003)). However, the object-design was very bad and not thread-safe. Therefore, this implementation is based on information from the man-page (see GNUPG (2002)), as well as from the accompanying documentation (see GNUPG (2003a)).

GnuPG can read all data from STDIN and can be configured by the `--status-fd` parameter to print all status information to STDOUT. Specifying `--batch` as another option allows complete control of GnuPG by other programs. For forcing encryption to all keys, whether they are trusted or not, for the specific operations the parameter `--always-trust` is specified, which turns off all trust checks GnuPG normally performs.

The `GPGCryptographicProvider` class uses two private methods for supporting all calls to the GnuPG's executable: runGPG and input2output

```
private void Process runGPG(String arguments)
```

[26] For more information about the GPL see FSF (2003)

```
private byte[] input2output(Process p, byte[] out, boolean waitForOut-
          put)
```

`RunGPG` takes a String as a parameter, which contains all options for calling GnuPG, appends the standard options `--batch` and `--status-fd 1` to the command line and spawns a process which is returned as the method's return value.

This process can be passed to the `input2output` method, which takes a process as the first argument, the data, which should be sent to the process, as the second argument and a boolean parameter, which specifies, whether the method should wait for the process' output, as the third parameter.

After it sends all the data to the process, it waits for the process to exit and checks the return code for success. If the process exited normally, all data is read from the process via a reader and parsed into lines. Normally, one would do this using a `BufferedReader`, however the `BufferedReader` can not determine, if more data is available, so it was necessary to implement this in this method manually. All lines starting with `[GNUPG:]` are discarded, because they are GnuPG's status output. All other data is appended to a temporary `StringBuffer` which is converted using the UTF-8[27] standard to a byte-array. This byte-array is the return value from this method. UTF-8 has been chosen because GnuPG's output is either UTF-8 or 7-Bit ASCII, which is compatible to UTF-8 for the represented character set.

`RunGPG` and `input2output` are used in conjunction by the methods `encrypt`, `encryptAndSign`, `decrypt` and `isValidSignature` with the corresponding parameters of GnuPG `-ea --always-trust`, `--eas --always-trust` and `--decrypt --always-trust`.

6.5.6 GPGKey

`GPGKey` is the implementation of the `Key` interface for use with the `GPGCryptographicProvider`. Internally, the `GPGKey` holds a reference to the full key-fingerprint from which the key-id can be extracted. Whenever a `GPGKey` is used in conjunction with the `GPGCryptographicProvider`, the provider extracts the key-fingerprint and passes it to the GnuPG executable to reference the key. Therefore, the key has to be imported into the local key-ring first, which is required by the `CryptographicProvider` interface and therefore presents no problem.

[27] UTF-8 is an encoding standard used to encode Unicode characters with one to three bytes, depending on the character. For the characters numbered 0 to 127 it is identical to the ASCII-standard

6.5.7 OpenPGPTransportPackager

The `OpenPGPTransportPackager` is an implementation of the suggested OpenPGP-Transport Security. It obtains the cryptographic provider for the OpenPGP standard from the `CryptographicManager`. Most likely, this will be the `GPGCryptographicProvider`. However, because of the loose coupling, one can introduce a new OpenPGP-compliant implementation by registering it with the `CryptographicManager`.

The `OpenPGPTransportPackager` receives the byte-array and writes it into an temporary file onto the disk. The temporary file resides in the global temporary directory, which normally should be cleaned up by the operating system. Afterwards, it opens the file using the ZIP routines provided by the Java™ class library. It extracts all files specified by the OpenPGP-Transport Standard. All file locations are put to the `AgentInfo` so that the agent can be packaged again.

The `OpenPGPTransportPackager` then uses the `SecurityPolicyManager` to let a security-policy be assigned to the agent. The resulting `AgentInfo` object is then returned to the caller.

6.5.8 OpenPGPTransportPassportReader

The `OpenPGPTransportPassportReader` implements the logic for reading the passports as already described in the OpenPGP-Transport Standard. It verifies the passport's signature and reads the security properties.

This passport-reader can be used by all packagers wanting to read this kind of passports.

6.5.9 AgentJarClassLoader

The received classes have to be loaded into the JVM. For this, Java™ defines the `Class-loader` class. A special classloader has been implemented for loading agents from their corresponding Java™ archive. The classloader shields implementation classes from the framework, so that the agent cannot directly access any implementation circumventing the security-policies in use.

For doing this, the classloader first tries to load classes from the given agent archive. Only if it cannot find the classes, it checks, if the agent may access the local class and then delegates the request to the system's class-loader.

As a template the example classloader by Ken McCrary (McCrary (2000)) was used and customized to the special requirements from loading classes of the agent archive first.

This classloader can be used by all packagers which need to load agents from Java™ archives.

6.6 Possible Improvements

The framework itself is consistent and very flexible, so that it is possible to implement all kinds of p2p-networks based on mobile agent technology on top of it. However, the basic implementations are, as the name suggests, only rudimentary implementations, which can be implemented better according to the scenario. This chapter shows specific problems of the implementations in certain situations and how they can be solved.

6.6.1 Defend against Denial of Service Attacks

"On the Internet, a denial of service (DoS) attack is an incident in which a user or organization is deprived of the services of a resource they would normally expect to have. Typically, the loss of service is the inability of a particular network service, such as e-mail, to be available or the temporary loss of all network connectivity and services." (Whatis?com (2003))

This means the purpose of a denial of service attack (DoS) is to disrupt the functionality of a system. The `BasicContainer` is vulnerable to this kind of attacks, because it can not limit the memory consumption of an agent. In theory, an agent can request all the memory available to the JVM thus blocking all other operations of the p2p-client, as shown in this example:

```
public void start()
{
     Vector v = new Vector();
     while(true)
     {
          // request 1 MB per turn
          v.add(new byte[1024*1024]);
     }
}
```

Another problem is an agent which will not cease its execution if it is told to do so, thus occupying threads and probably blocking the space for other agents, if the number of agents is limited on a peer. This can be accomplished by using a single endless loop:

```
public void start()
{
     while(true);
}
```

The latter problem has been addressed by simply killing the agent through the `Thread.stop`-method. Unfortunately, this method has been deprecated, because it may cause damage to the Sun Microsystems' ™ implementation of the JVM. There are other JVMs available, but the implementation of Sun Microsystems™ is the one most used. Furthermore, the deprecation status means, that the method may be removed in any future Java™ API specification without further notice.

A solution to this problem would be the use of a new JVM-instance for each agent. The JVM can be limited in memory consumption and can be killed like any normal task by the operating system. However, spawning a JVM for each agent is very slow. Furthermore, all management, like resource providers, network management etc. would reside in one thread, while the agents are running in others, so inter-process communication has to be used to communicate. Remote Method Invocation (RMI) or sockets can be used for this, but both are imposing further performance penalty and are complicating the implementation.

However, this can be avoided in scenarios, where code can be trusted. For instance, the `BasicContainer` can be used without problems, if it is possible to only accept code by a trustworthy programmer, e.g. the programmer's certificate is used.

Note that the problem exists in all method calls to an agent, not only in the start-method. For example during de-serializing an agent can block the system as well with an endless loop in its `writeObject` and `readObject`-methods.

7 Implementation Suggestions & Examples

7.1 Broad- und Multicasting of Messages

Broad- and multicasting can be implemented in two ways: The easiest solution is the use of a relay-agent, where other agents can get registered by sending a special message. Afterwards, registered agents can send specially prepared messages to the relay-agent, which in turn sends them to all other registered agents.

The other solution is the use of a virtual protocol. A new messaging protocol can be registered. This protocol is a virtual protocol that means that there is not real p2p protocol or local message delivering protocol associated with it. In fact, this protocol does not send any message on its own, but will forward all messages to the given groups participants. For example, it would be possible to create a virtual protocol, which is implemented in a way, that a message, addressed to broadcast:agents, will be sent to all local agents.

The first, agent-based approach is suitable for a dynamic group, that means, groups where other agents can join and which exists only for a short period. For example, this approach is very good for building discussion groups (for agents or humans) where interested parties can subscribe and after the meeting, the relay-agent dies. Furthermore, this approach works bidirectional for agents distributed over multiple peers, as long as they are allowed to send messages to remote agents.

The second, protocol-based, approach is suited to long-life groups or groups with fixed participants. It works more transparently than the agent-based approach. However, only local agents can send messages, because the protocol is only accessible on the same node. But relaying messages to remote agents is not a problem; that way, remote communication is only unidirectional.

7.2 Custom packagers

A custom packager has to implement the interface `.packaging.Packager`. This interface is very simple, because it only contains two method definitions for converting an agent into a byte-array and vice versa. However, the implementation of such packagers is even more cumbersome, because a packager has wide variations of responsibility: It has to implement the specified transport standard, extract and verify the class files, read

all security attributes, make security checks, install its own class loader and finally instantiate the agent.

Most of this functionality can be written quite modular and often existing implementations, which suit the needs for each subtask, can be used. For example, for reading an agent-passport, there is an interface `PassportReader` and there is already an implementation to read the passports used by the OpenPGP Transport. If the new standard uses the same passports, this passport-reader can be reused.

Cryptographic routines are commonly used by the custom packagers as well. The implementations should acquire a reference to cryptographic providers through the corresponding `CryptographicManager`, thereby reusing code and further decoupling the system.

The same applies to the classloaders: Packagers normally will have to read classes from Java™-archives. For this, they can use the `AgentJarClassloader` implementation found in the framework or use existent class loaders. The only thing a classloader should really take care of is the fact that the agent should not be allowed to load arbitrary classes. For example the framework's classloader blocks everything except java.lang and the framework's interfaces. During the application initialization, the new packager has to be registered with the `PackagingManager`. Afterwards it is usable by the p2p-network as well as the container etc.

7.3 Custom resource-providers

For every resource-type tat the agent wants to access, like files, databases etc., a resource-provider has to be registered in the system. The interface ResourceProvider must be implemented, which means methods for opening and closing resources have to be written. These methods must return `Resource` objects, which are used for actually accessing the data. For instance, to build a resource-provider for files, simply create a factory for the `Resources` where a `FileInputStream` or `FileOutputStream` is encapsulated. Note, that there are several methods to obtain data from a `Resource` object, like getting lines or a stream.

Implementing a resource-provider is therefore quite easy, because it normally only means wrapping existing APIs. However, it must be assured, that a read-only resource is really only readable, because otherwise the whole security concept does not work. For an example `HTTPResourceProvider` see the example application at the end of this document.

7.4 Peer-to-Peer-Network

Designing a well-scaling p2p-network implementation is a very difficult task. However, integrating it into the framework is quite easy. Normally one can develop the p2p-network independently, that means one will develop an implementation which can discover nodes, manage connections and send packages. The only thing to do is to build a wrapper to the `P2PNetwork` interface which defines more specific packet-types like one for a message or for an agent.

The `P2PNetwork` implementation has to send these packets over the network. On the other side, the implementation has to decide, whether it received a message or an agent.

If it received a message, it has to convert the message's receiver and sender from the network representation to the application's naming convention. Most notably, it has to use the local:-protocol for local entities. Afterwards, it can simply call the `Messaging-Manager` to send and route the message.

If it received an agent, the process is a little bit more complex: It has to put all the agent data into a byte-array. Afterwards it has to obtain a reference to the corresponding `Packager` object. This reference can be obtained by the `PackagingManager`. Note that it is necessary to specify the format the `Packager` should understand. This is necessary, because the format is either defined by the implemented p2p-network standard or it is transmitted with the agent data. Either way, the p2p-network implementation is the only instance which can decide which packager must be used. Using the packager, the agent can be unpackaged, resulting in an `AgentInfo` object. This object can be inserted into the container in use, which can be obtained via the `ContainerManager`. If the agent is running, the result or failure information can be transmitted back to the other peer in the network.

Furthermore, the network's peers have to be published to the application, allowing agents to decide, to which peer they want to travel. This is done by exposing objects, which implement the `Node` interface. Normally, the `Node` interface is implemented by an object that is only a wrapper around the real node's network address and forwarding method requests, like if a node is online etc., to the p2p-network implementation.

7.5 Itinerary protection scheme

For use in a more sophisticated application, the itineraries, i.e. the path the agent travels, can be protected via various methods. A very sophisticated one is part of the MARISM-A security-architecture (see Robles (2002)). MARISM-A is designed to secure the Jade

platform, but the same protection scheme can easily be implemented using this framework.

The protection scheme is based on a stub agent, which carries classes as instance data. The classes are encrypted to the corresponding host-keys. On every host the stub-agent determines on which host it runs and passes the encrypted class files data to a service. This service verifies if the right agent accesses the classes, by checking hash-values of the agent's code against one appended to the classes to be loaded. This way, it can be assured, that no one tampered with the agent in transit.

The implementation in this framework can be quite easy: The hash-values can be calculated by a custom packager during the receive process of the agent. The agent may request a new service, e.g. named `AgentClassLoaderService`, which is registered in the system through the `ServiceManager`, to decrypt the classes and return a new class loader for them. In turn, The `AgentClassLoaderService` can check the agent's passport for the hash-values and is therefore able to check the decryption attempt.

7.6 Offline Applications

With the rise of new, small and mobile devices connecting to the Internet, new applications are possible, like initiating searches for accommodation, traffic information etc., if someone is on the road. However, traffic and connection fees are still very high, so that it is necessary to reduce the amount of data sent from and to the devices and the time the devices are connected to the network. Sometimes it is even necessary or likely that a device or another node is not connected all the time to the network which is needed to accomplish the tasks. In either case, mobile agents can be used to develop applications, where the initiating node does not need to be online all the time. Instead, the owner sends a mobile agent, which roams the network for the desired information. After all information is collected, there are two possible implementations, which are applicable under different scenarios:

1. The agent waits on a node somewhere in the network until the initiating node comes online again. If the node, where the agent waits, goes down, the agent has to look for another node to wait. However, it might be to slow to do this or for other reasons, the node goes down without notifying the agent and all information is lost.

2. The agent stops at a node, where a special resource-provider is installed. This provider offers a persistent storage for agents. For example, it can offer 500 Kbytes to certified agents. This way, an agent may request a resource like agent-

data://somename. The `ResourceProvider`, for example named `AgentData`, has to be registered before with the `ResourceManager`. This resource-manager will then be used to provide the resource, if the security-policy for the agent allows the use of this resource-type. The agent gets a reference to a `Resource` object and can use it to write its results to. These results can be fetched by the initiating node via another agent or by special messages exchanged between the nodes. If the storage is always online, like it is provided by a service provider, this method is safer than the first one. However, it needs write access to resources, which may be a problem in public networks.

7.7 Agent-Supported Download Applications

P2P-File-sharing-applications are quite popular today. However, they depend on searching the network via broadcasts and trusting the unauthenticated and invalidated results, which are coming back. By using mobile agents, it is possible to verify the results and only send them back, if they really match the search results or other given criteria.

Therefore, they can travel to the node, which offers that source, read the data and validates it. Afterwards it needs to send the data to its owner. This can be accomplished by giving an agent a unique job-number. When the agent is initiated, it receives this job-number. Furthermore, a `ContainerMessageListener` is registered with the `Container` object in use, which intercepts all messages and saves the contents of those, who start with the job-number, to an associated file. The agent can therefore simply send the verified data to its owner's host-address including the job-number, so that the owner’s host is able to handle the incoming data.

8 Anonymizer Sample-Application

8.1 Application's focus

As part of this project, a sample application is provided, which utilizes the framework for implementing an anonymizing service, which is based on mobile agents. This way the functionality and the design of the framework can be demonstrated.

The sample application's purpose is to anonymize web requests, issued by an ordinary web browser. This is done via mobile agents, which are issued for each request and are roaming the p2p-network to anonymously fetch the requested data for the user.

8.2 Anonymizing traffic

Today more and more people are using the Internet, especially the World Wide Web (WWW). Many things are conducted digitally, including e-commerce and information gathering. However, each packet of data sent in the Internet contains the sender’s unique address. Defined in the Internet Protocol (IP) this address is a 32 bit number which is assigned to a computer and every other device connected to the Internet. Because most computers are only used by one person at a time, the IP address not only correlates to the computer but also to the user.

This way it is possible to create user profiles by analysing the generated traffic. For example website operators are able to analyse logs and secret services are capable of capturing and analysing networking traffic in central routing points. This practice conflicts with the right of privacy that citizens in modern democracies enjoy.

To re-establish privacy in the digital era, traffic needs to be anonymized. This is often done by sending it randomly through the network, getting relayed at each computer it passes. Thereby the original source address can be obscured because each computer retransmits the packet and thereby places its IP address in the packet.

Systems working like this are called mixnets. Mixnets are networks, in which messages are routed randomly between the nodes to obscure the senders’ original addresses by making traffic analysis harder. Since each node only knows from

However, issuing a request anonymously is not as difficult as returning the request's result anonymously as well, because for this the request's source needs to be known and therefore needs to be stored somewhere.

8.3 Application flow

The request from the web-browser is sent to the proxy-server which is a node in the p2p-network at the same time. For each page or graphic that the browser requests, a request-id is generated. Afterwards the connection is saved with this id. A new agent is spawn which gets the request-id and the URL it should fetch. Upon start-up the agent initializes a list of nodes it travelled to. This list contains a random-length initialisation, which consists of valid host-names in the p2p-network. Since this list has only valid names and has a random length, the real origin of the agent cannot be determined by looking on this list.

The agent then proceeds by visiting a randomly chosen number of hosts. At each host, it adds the host-name to its host-list.

On the last node the agent issues the request for the URL it has to fetch. The data are stored within the payload reference and the agent will go back its internal host-list, getting the previous host, it has visited and will request migration to that host. On each host the agent will try to deliver the data. This is done by requesting a resource named return://requestid. If the resource is found, the request has been issued by that host and the agent will write the data to the obtained resource. The stream of the resource is directly connected to the socket of the connection to the browser. This way the agent will write its response directly to the browser.

If the resource could not be found, the agent travels back down the list until it reaches its origin. If one host is down, it will be skipped and the agent will proceed to the previous one in the list. This way, it does not matter, if one host in the list is down, unless it is the originating node. If this happens, the agent will reach the list's end and will quit execution.

8.4 Application's implementation

8.4.1 Package structure

The package structure of the anonymizer-application is based on the framework's structure. The main package for the application is

`de.tuclausthal.informatik.winf.anonymizer`.

If sub-packages to this package are referenced in the following, a dot "." is prefixed, that means:

`.packaging` is `de.tuclausthal.informatik.winf.anonymizer.packaging`.

The packages `.agent`, `.p2p`, `.packaging` and `.resource` hold the implementation to the interfaces in the corresponding packages of the framework. In the `.gui`-package the complete graphical user interface (GUI) for the anonymizer is stored. The `.proxy`-package contains the implementation of the proxy-server part of the application.

8.4.2 Proxy-Server

A proxy-server is a server which accepts requests from clients and tries to fetch the given data. It operates on the application-layer of the ISO/OSI layer model[28] and is therefore able to understand the data transferred from and to the client. Because of this, it is possible to filter content, for example block sex sites within a company. Another advantage of proxy-servers is that requests can be cached and then served from the cache, if multiple clients access the same resources, like web pages.

A proxy-server is configured within the browsers for proxy-servers serving web-pages via the hypertext transfer protocol (HTTP). By pointing the browser to the anonymizer, the anonymizer is able to know which page should be fetched.

The proxy-server's implementation is very simple: One object of the built-in class `ServerSocket` is instantiated and bound to port 8080. The incoming requests by a web-browser are then accepted. HTTP requests are looking like this:

```
GET http://www.tu-clausthal.de/ HTTP/1.0
Accept: image/gif, image/x-xbitmap, image/jpeg, image/pjpeg
Accept: application/vnd.ms-powerpoint, application/nvd.ms-excel
Accept: application/msword
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
```

From this whole request only the unified resource locator (URL) of the first line is used and passed over to a new agent. In this case, this is http://www.tu-clausthal.de/. All other information, like browser identification or cookies, are discarded, because they can be used to track the user's session. Along with the URL the agent gets a newly generated ID, which is used to identify the connection with the browser. This connection is stored inside a `HashMap` and can be retrieved using the connection's ID.

[28] The ISO/OSI model is described by Tanenbaum (Tanenbaum (2000))

Because spawning a new agent is not much slower than spawning a new thread and the proxy-server will only be accessed by one user, the server itself is not multi-threaded.

The proxy-server's source is located in the class `ProxyServer` in the package `.proxy`.

The proxy-server registers a new resource-provider in the system, which handles all URLs of the type return. This resource-provider is implemented via an inner-class.

8.4.3 Anonymizing Agent

The request is handed over from the proxy-server to an agent. The agent is responsible to fetch the data. For this, the agent initializes itself at the first start. To determine, if initialization is necessary, the agent checks its instance variable mustInit. If this variable is true, the agent initializes itself, by proceeding in its `init`-method:

```
protected void init(AgentServices agentServices)
{
     if (!this.mustInit)
          return;

     // random route
     String[] neighboringNodes = agentServices.getNeighborNodes();

     if (neighboringNodes != null && neighboringNodes.length > 0)
     {
          this.hopsToGo = (int) (Math.random() * MAX_HOPS) + 1;

          // init with fake data
          int fakeCount = (int) (Math.random() * 30);
          for (int i = 0; i < fakeCount; i++)
          {
               this.hosts.add(
                   neighboringNodes[(int)
                         (Math.random() * neighboringNodes.length)]);
          }
     }
     else
     {
          // rarely happens: no neighbors
          this.hopsToGo = 1;
     }

     // last entry ever is our host
     this.hosts.add(agentServices.getNodeName());
     this.mustInit = false;
}
```

This method constructs a faked initial travel-list for this agent to obscure the real owner. To get valid host-names, a list containing all neighbour-nodes is requested from the system. Names from this list are taken to put a random number of fake-entries at the top of

the list. Additionally the `hopsToGo` instance-variable is initialized with a random value, which must not exceed the compile-time constant `MAX_HOPS`. If no neighbouring peers are available, the agent falls back to unanonymized mode. That means that the `hopsToGo`-field is simply initialized with 1.

After the initialization is guaranteed the agent determines if it is in forward- or in backward-movement by looking at its payload-data.

If the payload is a `null`-reference, the agent is in forward-movement and checks, if it has to go to further hosts. If it must, it chooses one random neighbour host, adds it to its list and travels there. It does so until it successfully migrates to another host. This behaviour is implemented in the `moveForward`-method and called from the agent's `start`-method. If the agent has to get the data, because the `hopsToGo`-field is zero, the agent requests the data by acquiring the corresponding resource via its `AgentServices`' instance and saves it in a field named `payload`. This is implemented in the method named `readData`.

In backward-movement the agent tries to deliver its data on the current host. This is done by requesting a resource with the given request-id. If this fails the agent chooses the next host from its list and migrates there and proceeds. If the resource could be opened successfully, the agent writes its entire payload to that resource and terminates. The backward-movement is implemented in the method `moveBackward` and the attempts to deliver the data in the method named `deliverData`.

8.4.4 Peer-to-Peer-Implementation

The p2p-network implementation is designed to support static p2p-networks that means that nodes neither join the network nor leave it. All peers should be available at all time. Because of this assumption, the implementation is very clean and well-suited to demonstrate how to implement a p2p-network.

The p2p-implementation is available in the `SimpleP2P` class and the corresponding `SimpleP2PNode` class in the `.p2p`-package.

When going online, the p2p-network opens a socket on port 4444 using a `ServerSocket` object. Incoming connections will be handled via a separate thread. The thread's logic is implemented in the `run`-method of the SimpleP2P class. The `SimpleP2P`-protocol is thereby defined as following:

The first transmitted byte contains the command. The command codes can be seen in table 8-1.

Code	Command
0x00	COMMAND_PING: Requests that the other node answers with OK. All data sent with this command have to be discarded.
0x01	COMMAND_AGENT: An agent will be sent in the data and should be unpackaged and run.
0x02	COMMAND_MESSAGE: Reserved for future support of message-passing over the P2P-network.

Tab. 8-1 Command-codes of the Simple-P2P networking protocol

After the command, four bytes are transferred for the data length. Afterwards the data accompanying the command and are therefore command-specific are sent. For each command the result code is returned to the initiator. Afterwards the connection is closed. The incoming data is read according to the protocol by the p2p-network implementation and actions are taken depending on the command. The command's result is then sent back to the sender. It may be 0x00 for OK and 0x01 for indicating an error.

For example, agents are unpackaged or sent to another peer. For each command a new connection is opened which is closed afterwards.

8.4.5 Agent Migration

Agents are migrated by only sending their state to improve the overall system's speed. Since only one kind of agent type exists in the system, this is sufficient and reduces the amount of data which is sent over the network significantly. To achieve this, a new packager has been implemented, which is called `StateOnlyPackager`. This packager simply stores the class name of the agent to instantiate and its serialized state into a large package, which can then be used to restore the agent on another peer. The `StateOnlyPackager` is derived from the abstract class `PackagerBase` from the framework, which offers routines for serializing the state of agents, so that the implementation is quite easy and straightforward. The unpackaging for example is completely done in the following method:

8.4.6 Resource-Provider for HTTP

The agents need to fetch the web pages. For this HTTP is used. To offer the agents the access to web-pages, a new resource-provider has been included in the application,

which handles HTTP requests. The agent simply opens a URL, like http://www.server.com, and gets a resource as a return value. This resource contains all of the HTTP response, including the header. That way, this data can be send to the browser when the agent delivers the data without any conversion. The resource-provider itself is very simple, since it only works as a factory for HTTPResource objects. The main work is done within the class `HTTPResource`: It opens a connection to the server specified in the URL and requests the data. This implementation does not use the Java™ HTTP handling classes but instead uses its own socket to preserve the original HTTP header sent by the server. The data is fetched in the constructor call:

```
HTTPResource(String url)
{
     this.url = url;

     try
     {
          // only for parsing
          URL tempURL = new URL(url);

          // get port
          int port = tempURL.getPort();
          if (port < 0)
                  port = 80;
          // get port to computer
          this.socket = new Socket(tempURL.getHost(), port);
          // encapsulate response
          this.inputStream = this.socket.getInputStream();

          // write HTTP Command
          OutputStream out = this.socket.getOutputStream();
          out.write(
               ("GET "
                  + tempURL.getPath()
                  + "?"
                  + tempURL.getQuery()
                  + " HTTP/1.0\nHost:"
                  + tempURL.getHost()
                  + "\n\n")
                  .getBytes());

          // wait for all data to arrive
          int data = 0;
          ByteArrayOutputStream tempOut = new ByteArrayOutputStream();
          while(data != -1)
          {
               data = -1;
               try
               {
                  if(this.inputStream.available() > 0)
                  {
                        byte[] buf = new
          byte[this.inputStream.available()];
                        this.inputStream.read(buf);
                        tempOut.write(buf);
                  }
                  data = this.inputStream.read();
```

```
                } catch(IOException e) {}

                if(data != -1) tempOut.write((byte)(data & 0xFF));
           }
           this.inputStream = new ByteArrayInput-
           Stream(tempOut.toByteArray());
      }
      catch (Throwable e)
      {
           e.printStackTrace();
           return;
      }
}
```

The HTTP resource-provider only supports HTTP GET commands. That means that most web-forms will not work with the anonymizer. However, such forms often require the authentication of the user, so this is not a big disadvantage.

8.5 Application Usage

8.5.1 Configuring the network

The p2p-network is statically configured and initialized during the start-up of the application. This implementation is not able to determine its neighbours automatically.

Therefore, all peers have to be entered into a text-file in the applications working directory. The file is named hosts.txt and contains one host per line. The host can be given either as an IP address in the form aaa.bbb.ccc.ddd (e.g. 192.168.1.1) or as a DNS name (e.g. computer.domain.com). All lines starting with a hash-sign (“#”) are considered as comments and are ignored. An example hosts.txt file can look like this:

```
# hosts.txt
# part of the Anonymizer
192.168.54.1
mickey.intranet.daniel-luebke.de
neo.intranet.daniel-luebke.de
```

Note, that the application reads this file during start-up. If the user wants to edit his neighbour-list he has to stop and restart the anonymizer.

8.5.2 Starting and using the application

The anonymizer-application can be started by using the JRE with the following command:

```
java -jar anonymizer.jar
```

This tells the JRE to launch the application packaged in the archive named anonymizer.jar. The application will automatically start all services and presents a GUI, in which he is able to control and monitor his p2p-client.

The GUI is organized within four tabs, which can be seen in figure 8-3.

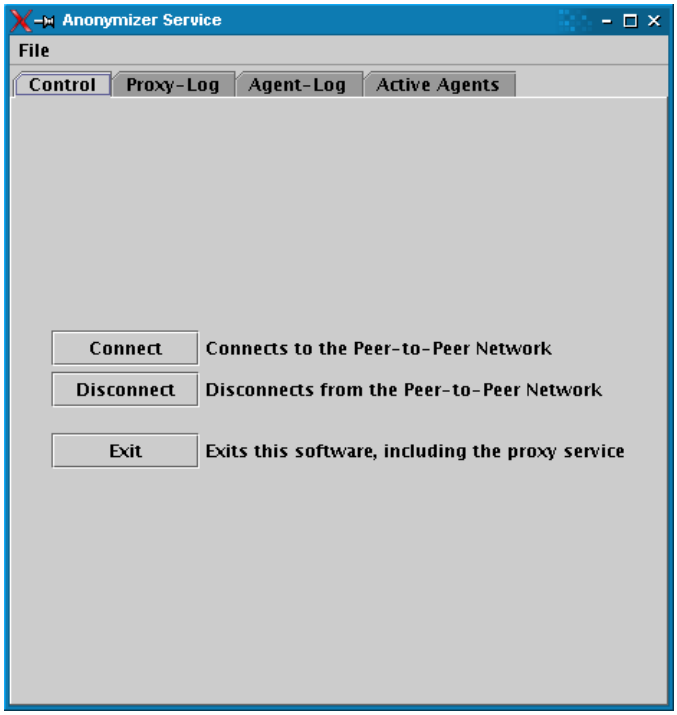

Figure 8-1: The anonymizer application

The first tab has buttons to control the software: It allows the connection and disconnection from the network.

The second tab contains the proxy-log. The proxy-log lists all URLs which have been requested and served by this p2p-node.

The third tab displays the agents' status-changes. Whenever an agent starts or stops within the local system, this change will be logged here.

The last tab shows, which agents are currently running on the system. This can be used to see the load on the system. To not negatively affect the performance by refreshing this display, a refresh-button is placed at the bottom of this tab. By pressing this button, the user can refresh the list of actively running agents on the system.

By closing this window or choosing “Exit” from the menu “File”, the application can be terminated, including the shutdown of all network activity.

9 Conclusions and Outlook

The objectives of this project are to provide a framework, which is a solid foundation for the development of an open p2p-network. The final p2p-network should be powered by mobile agents to distribute logic and avoid frequent updates of software. It has to incorporate strong security-measures to defend against attacks concerning the privacy and the data of the users.

Therefore, the first step was to develop a security model which can protect agents and their state in an open network. The resulting OpenPGP Transport Standard is very flexible and allows the inclusion of further security properties if necessary through the use of an agent-passport. For being useful in completely decentrally organized networks or within environments where autonomity is necessary, the key-validation process is done decentrally allowing its use in such cases. This has been achieved by using the OpenPGP standard for encryption and digital signatures.

The second step was to develop the framework which will capture the design for the client-software of the p2p-network. This objective was solved by gathering the requirements of the applications based upon the framework. Afterwards, these requirements where transferred into a complete framework-design. The resulting framework has a very flexible design, in which functionality domains are independent of each other. Because of this it is no problem to independently develop parts of the p2p-network and put them together afterwards. Some further ideas for applications and specific problems and how to solve them using the framework were given to complete the framework's description.

The third objective was to provide a sample application. This application has demonstrated that the framework design can be used to efficiently implement agent-powered p2p-applications with minimal effort. Furthermore, the flexibility of the design has been shown by replacing the packager and inserting a new p2p-network implementation without any modifications of the system. The sample application is important for another reason as well: It shows how to gain anonymity within applications based on the framework. The anonymity is independent of the p2p-implementation because all of the logic is implemented in the agent. All that is required is that the used packager does not place properties in the transferred agents by which the owner can be identified.

Finally, the sample application demonstrates that this framework is a solid foundation. Future development should concentrate on a scalable p2p-implementation and should deal with the defence against denial-of-service attacks.

P2P-implementations of other projects can possibly be re-used for this. For example, Sun Microsystems™ offers the JXTA framework for p2p-applications. Other popular protocols like the Gnutella or the Freenet[29] protocol could be extended to support mobile agents as well. A new implementation based on newer research results could be used to compare performance.

After developing and including the best scalable p2p-implementation, the resulting network will be able to allow the safe use of distributed resources and will offer unmatched functionality and use-cases. This is possible because each user can insert his own logic into the network by writing and deploying his own agents. Examples and possible scenarios were given in the seventh chapter. It is possible to move all of today's existent p2p-services to the new network, including instant messaging, file-sharing and distributed computing.

Note that for interoperability the framework supports multiple p2p-implementations to connect to, so that beside the main-protocol the software will be able to connect to other networks as well.

Further research is needed in the areas of security. This especially includes dealing with malicious hosts in an open p2p-network and replay-attacks which are not yet covered by the security mechanisms presented in this work. While the problem of malicious hosts has not been solved even in closed networks, it is unlikely that a solution will be found soon. Because of this, actions limiting the impact of malicious hosts would be interesting to investigate and implement. The latter problem of replay-attacks can be solved by including serial numbers. Further approaches have been developed for predefined itineraries and can possibly be ported to the framework and the open network environment.

An implementation of a container, which is capable of further restricting the agents by spawning new JVMs instead of threads, would be interesting in terms of performance and reliability. This type of implementation could be used to defend against DoS-attacks.

All in all the combination of mobile agents and p2p-networks is a very interesting approach for enhancing today's networks and integrating different services into one common platform. The framework developed in this project provides a solid foundation which should be used to implement the mentioned functionality and can be used for further research to resolve the open questions.

[29] The Freenet project home page is http://freenet.sourceforge.net/

References

Deutsch, L. Peter (1989): Design reuse and frameworks in the Smalltalk-80 system, in: Software Reusability, Volume II: Applications and Experience, pp. 57-71, Addison-Wesley, Reading.

Foster, I. and Iamnitchi, A. (2003): On death, taxes, and the convergence of peer-to-peer and grid computing, 2nd International Workshop on Peer-to-Peer Systems (IPTPS'03).

Gamma et. al. (2001): Design Patterns – Elements of Reusable Object-Oriented Software, 22nd printing, Addison-Wesley, Reading.

GNUPG (2003a): Format of colon listing, shipped in the file DETAILS.gz with GNUPG.

Grand (1998): Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML, John Wiley & Sons, Indianapolis.

Jennings, Nicholas R.; Wooldridge, Michael (2000): Agent-Oriented Software Engineering, Proceedings of the 9th Europeans Workshop on Modelling Autonomous Agents in a Multi-Agent World: Multi-Agent System Engineering (MAAMAW-99).

Johnson, Ralph E. and Foote, Brian (1998): Designing reusable classes, Journal of Object-Oriented Programming June/July 1998.

Loo, Alfred W. (2003): The Future of Peer-to-Peer Computing, Communications of the ACM, September 2003/Vol.46, pp. 57-61.

Lübke, D. and Marx Gómez, J. (2003): Designing a Framework for Mobile Agents in Peer-to-Peer-Networks, Proceedings of SCI 2003.

Lübke, D. and Marx Gómez, J. (to be published in 2004): Usage of OpenPGP with mobile agents in p2p-networks, Proceedings EIS 2004, Workshop on intelligent Agents in Peer-to-Peer-Networks.

McConnell, Steve (1993): Code Complete, Microsoft Press, Redmond.

Oestereich, Bernd (2001): Objektorientierte Softwareentwicklung – Analyse und Design mit der Unified Modeling Language, Oldenborg Wissenschaftsverlag, Munich, 2001.

Pang, X.; Catania, B.; Tan, Kain-Lee (2003): Security Your Data in Agent-Based P2P Systems, Proceedings of Eighth International Conference on Database Systems for Advanced Applications (DASFAA '03).

Bierman, Elmarie; Cloete, Elsabe (2002): Classification of malicious host threats in mobile agent computing, ACM International Conference Proceeding Series: Proceedings of the 2002 annual research conference of the South African institute of computer scientists and information technologists on Enablement through technology.

Rivest, R.; Shamir, A. and Adleman, L. (1978): A Method for Obtaining Digital Signatures and Public-Key Cryptosystems, Communications of the ACM, Volume 21 (2), pp. 120-126.

Robles, S.; Mir, J.; Borrell, J. (2002): MARISM-A: An Architecture for Mobile Agents with Recursive Itinerary and Secure Migration, In 2nd. IW on Security of Mobile Multiagent Systems, Bologna.

Tanenbaum, Andrew S. (2000): Computernetzwerke, Addison-Wesley, Reading.

Tanenbaum, Andrew S.; Stehen, Maarten van (2003): Verteilte Systeme, Pearson Studium.

Lexica

Hyperdictionary, 2003, http://www.hyperdictionary.com, especially
http://www.hyperdictionary.com/computing/one-way+hash+function
http://www.hyperdictionary.com/computing/java+archive.

Whatis?com, 2003, http://www.whatis.com, especially
http://whatis.techtarget.com/definition/0,289893,sid9_gci213591,00.html.

Internet Addresses

ACCC (2000): Figure 2: Asymmetric or Public Key Encryption,
http://www.uic.edu/depts/accc/newsletter/adn26/figure2.html, 2003-11-25.

Barkstrom (2000): The Standard Waterfall Model for Systems Development, http://asd-www.larc.nasa.gov/barkstrom/public/The_Standard_Waterfall_Model_For_Systems_Development.htm, 2003-11-25.

Borland (2003): JBuilder, http://www.borland.com/Jbuilder/, 2003-11-25.

Callas, J.; Donnerhacke, L.; Finney, H.; Thayer, R. (1998): RFC 2440 – OpenPGP Message Format, http://www.rfc-editor.org/rfc/rfc2440.txt, 2003-11-25.

CCD (2003): Secure Sea-of-Data applications coming true… MARISM-A,
http://www.marisma.org, 2003-11-25.

CNET (2003): peer-to-peer network – Glossary,
http://www.cnet.com/Resources/Info/Glossary/Terms/peer.html, 2003-11-25.

Cooper (1998): The Design Patterns Java Companion,
http://www.patterndepot.com/put/8/JavaPatterns.htm, 2003-11-25.

Darthmouth (2003): D'Agents: Mobile Agents at Dartmouth College,
http://agent.cs.dartmouth.edu/, 2003-11-25.

DFN (2003): DFN S2S – Übersicht, http://s2s.neofonie.de/index.jsp, 2003-11-25.

DFN Cert (2003): DFN-PCA, http://wwwkeys.de.pgp.net, 2003-11-25.

Dierks, T.; Allen, C. (1999): RFC 2246 – The TLS Protocol, http://www.rfc-editor.org/rfc/rfc2246.txt, 2003-11-25.

DI Management (2003): RSA Algorithm, http://www.di-mgt.com.au/rsa_alg.html, 2003-11-25.

Dominus (2002): "Design Patterns" Aren't, http://perl.plover.com/yak/design/, 2003-11-25.

Eastlake, D.; Jones, P. (2001): RFC 3174 – US Secure Hash Algorithm 1 (SHA1), ftp://ftp.rfc-editor.org/in-notes/rfc3174.txt, 2003-11-25.

Eclipse Consortium (2003): Eclipse.org Main Page, http://www.eclipse.org, 2003-11-25.

eDonkey (2003): eDonkey2000 – Overnet, http://www.edonkey.com, 2003-11-25.

Erkomaa (1998): Secure Socket Layer and Transport Layer Security, http://www.tml.hut.fi/Studies/Tik-110.350/1998/Essays/ssl.html, 2003-11-25.

FCIT (1999): Florida Center for Instructional Technology: Software, http://fcit.coedu.usf.edu/network/chap6/chap6.htm, 2003-11-25.

FIPA (2002): FIPA Abstract Architecture Specification, http://www.fipa.org/specs/fipa00001/SC00001L.html, 2003-11-25.

FIPA (2002a): FIPA ACL Message Structure Specification, http://www.fipa.org/specs/fipa00061/SC00061G.html, 2003-11-25.

FIPA (2003): Welcome to the Foundation for Intelligent Physical Agents, http://www.fipa.org, 2003-11-25.

Fontana, John (2002): P2P getting down to some serious work, http://www.nwfusion.com/news/2002/0819specialfocus.html, 2003-11-25.

Freenet (2003): The Freenet Project, http://freenet.sourceforge.net/, 2003-11-25.

FSF (2003): GNU General Public License – GNU Project, http://www.gnu.org/copyleft/gpl.html, 2003-11-25.

Gentleware (2003): Poseidon for UML, http://www.gentleware.com, 2003-11-25.

GNUPG (2002): gpg, http://www.gnupg.org/(en)/documentation/manpage.en.html, 2003-11-25.

GNU (2003): GNU C Library, http://www.gnu.org/software/libc/libc.html, 2003-11-25.

GNUPG (2003): The GNU Privacy Guard, http://www.gnupg.org, 2003-11-25.

Goetz, Brian (2002): Java theory and practice: I have to document THAT?, http://www-106.ibm.com/developerworks/java/library/j-jtp0821.html?loc=j, 2003-11-25.

Gutman, Peter (2000): X.509 Style Guide, http://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt, 2003-11-25.

Harris (2002): PGP Keys with Duplicate Key IDs, http://skylane.kjsl.com/~jharris/duplicate_keyids.html, 2003-11-25.

Harris (2003): keyanalyze Results, http://keyserver.kjsl.com/~jharris/ka/, 2003-11-25.

Heise (2003): Krypto-Kampagne, http://www.heise.de/security/dienste/pgp/, 2003-11-25.

Horton (2003): New Tricks With Design Pattern: Java Value Types, http://javaboutique.internet.com/tricky_des_pat/, 2003-11-25.

Housley, R. W.; Ford, W.; Solo, D.(2002): RFC 3280 – Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile, ftp://ftp.rfc-editor.org/in-notes/rfc3280.txt, 2003-11-25.

ICQ (2003): Get ICQ instant messenger, chat, people search and messaging service!, http://www.icq.com, 2003-11-25.

Johnson, Don B. (1999): ECC, Future Resiliency and High Security Systems, http://www.nullify.org/docs/ECCFut.pdf, 2003-11-25.

Kartmann, Emmanuel (2003): Gnu Privacy Guard for .NET [v1.0], http://www.codeproject.com/csharp/gnupgdotnet.asp, 2003-11-25.

Limewire (2003): limewire.org, http://www.limewire.org, 2003-11-25.

Magee (2003): Software Engineering-Methods, https://www.doc.ic.ac.uk/~jnm/se_third_yr/patterns2-notes.pdf, 2003-11-25.

Marques, Paulo; Fonseca, Raul; Simões, Silve; Luís, João G. (2002): A Component-Based Approach for Integrating Mobile Agents Into the Existing Web Infrastructure, http://citeseer.nj.nec.com/marques02componentbased.html, 2003-11-25.

McCrary, Ken (2000): Create a custom Java 1.2-style ClassLoader, http://www.javaworld.com/javaworld/jw-03-2000/jw-03-classload.html, 2003-11-25.

McDowell (2002): Experimental PGP key path finder, http://the.earth.li/~noodles/pathfind.html, 2003-11-25.

Microsoft (2001): ActiveX: Developing Exciting Content and Applications fort he Internet and Intranets, http://msdn.microsoft.com/archive/en-us/dnaractivex/html/msdn_activexwp.asp, 2003-11-25.

Microsoft (2003): Microsoft Patterns, http://msdn.microsoft.com/architecture/patterns/MSpatterns/default.aspx, 2003-11-25.

Monday (2002): Java design patterns 201, http://www-106.ibm.com/developerworks/java/edu/j-dw-javapatt2-i.html, 2003-11-25.

PGP Corporation (2003): PGP Corporation, http://www.pgp.com, 2003-11-25.

JXTA (2003): jxta.org, http://www.jxta.org, 2003-11-25.

Ramsdell, B. (1999): RFC 2633 – S/MIME Version 3 Message Specification, 2003-11-25.

Redshift Research (2003): P2P User Statistics, http://www.redshiftresearch.com/P2PUserStats.asp, 2003-11-25.

Rivest, R. (1992): RFC 1321 - The MD5 Message-Digest Algorithm, ftp://ftp.rfc-editor.org/in-notes/rfc1321.txt, 2003-11-25.

Riehle, Dirk and Züllighoven, Heinz (1996): Understanding and Using Patterns in Software Development, http://citeseer.nj.nec.com/cache/papers/cs/5127/http:zSzzSzwww.ubs.comzSzezSzindexzSzaboutzSzubilabzSzprint_versionszSzpszSztapos-96-survey.pdf/riehle96understanding.pdf, 2003-11-25.

Sun Microsystems (1999): Naming Conventions, http://java.sun.com/docs/codeconv/html/CodeConventions.doc8.html, 2003-11-25.

Sun Microsystems (2003): The Source for Java Technology, http://java.sun.com, 2003-11-25.

Sun Microsystems (2003a): Applets, http://java.sun.com/applets/, 2003-11-25.

Sun Microsystems (2003b): Welcome to NetBeans, http://www.netbeans.org, 2003-11-25.

TC TrustCenter (2003): TC TrustCenter, http://www.trustcenter.de, 2003-11-25.

Tigris.org (2003): argouml.tigris.org, http://argouml.tigris.org, 2003-11-25.

Shirky, Clay (2000): What is P2P… And What Isn't? http://www.openp2p.com/pub/a/p2p/2000/11/24/shirky1-whatisp2p.html, 2003-11-25.

Verisign (2003): VeriSign Inc., http://www.verisign.com, 2003-11-25.

Watson (2003): The Delphi VCL (Visual Component Library), http://www.win.tue.nl/~watson/2R080/downloads/Sheets_Delphi_VCL.doc, 2003-11-25.

WOWN (2003): World of Windows Networking, http://www.wown.com, 2003-11-25.

Youd, David (2000): What is a Digital Signature?, http://www.youdzone.com/signature.html, 2003-11-25.

Zorgdrager (2001): Developing Java solutions using Design Patterns, http://www-106.ibm.com/developerworks/ibm/library/it-kelby1/, 2003-11-25.